PRECISIONSM RESOLUTION TARGETS

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file Votre référence

Our file Notre référence

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Bulk Synchronous Parallel: Practical Experience with a Model for Parallel Computing

by

Anton Saarimaki

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University

Ottawa, Ontario

March 1995

© Copyright

1995, Anton Saarimaki

National Library
of Canada

Bibliothèque nationale
du Canada

Acquisitions and
Bibliographic Services Branch

Direction des acquisitions et
des services bibliographiques

395 Wellington Street
Ottawa, Ontario
K1A 0N4

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file - Votre référence

Our file - Notre référence

THE AUTHOR HAS GRANTED AN
IRREVOCABLE NON-EXCLUSIVE
LICENCE ALLOWING THE NATIONAL
LIBRARY OF CANADA TO
REPRODUCE, LOAN, DISTRIBUTE OR
SELL COPIES OF HIS/HER THESIS BY
ANY MEANS AND IN ANY FORM OR
FORMAT, MAKING THIS THESIS
AVAILABLE TO INTERESTED
PERSONS.

L'AUTEUR A ACCORDE UNE LICENCE
IRREVOCABLE ET NON EXCLUSIVE
PERMETTANT A LA BIBLIOTHEQUE
NATIONALE DU CANADA DE
REPRODUIRE, PRETER, DISTRIBUER
OU VENDRE DES COPIES DE SA
THESE DE QUELQUE MANIERE ET
SOUS QUELQUE FORME QUE CE SOIT
POUR METTRE DES EXEMPLAIRES DE
CETTE THESE A LA DISPOSITION DES
PERSONNE INTERESSEES.

THE AUTHOR RETAINS OWNERSHIP
OF THE COPYRIGHT IN HIS/HER
THESIS. NEITHER THE THESIS NOR
SUBSTANTIAL EXTRACTS FROM IT
MAY BE PRINTED OR OTHERWISE
REPRODUCED WITHOUT HIS/HER
PERMISSION.

L'AUTEUR CONSERVE LA PROPRIETE
DU DROIT D'AUTEUR QUI PROTEGE
SA THESE. NI LA THESE NI DES
EXTRAITS SUBSTANTIELS DE CELLE-
CI NE DOIVENT ETRE IMPRIMES OU
AUTREMENT REPRODUITS SANS SON
AUTORISATION

ISBN 0-612-03008-3

Canada

Name ANTON SHARIMAKI

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

COMPUTER SCIENCE

SUBJECT TERM

1774

U·M·I

SUBJECT CODE

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0723
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0279
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language 0679
General 0289
Ancient 0290
Linguistics 0291
Modern 0401
Literature 0294
General 0295
Classical 0297
Comparative 0298
Medieval 0316
Modern 0591
African 0305
American 0352
Asian 0355
Canadian (English) 0593
Canadian (French) 0311
English 0312
Germanic 0315
Latin American 0313
Middle Eastern 0314
Romance 0501
Slavic and East European 0503

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion 0318
General 0321
Biblical Studies 0319
Clergy 0320
History of 0322
Philosophy of 0469
Theology 0323

SOCIAL SCIENCES

American Studies 0323
Anthropology 0324
Archaeology 0326
Cultural 0327
Physical 0310
Business Administration 0272
General 0770
Accounting 0454
Banking 0338
Management 0385
Marketing 0501
Canadian Studies 0503
Economics 0505
General 0508
Agricultural 0509
Commerce Business 0510
Finance 0511
History 0358
Labor 0366
Theory 0351
Folklore 0578
Geography 0460
Gerontology 0383
History 0386
General 0386

Ancient 0579
Medieval 0581
Modern 0582
Block 0328
African 0331
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0585
History of Science 0398
Law 0615
Political Science 0616
General 0617
International Law and Relations 0814
Public Administration 0452
Recreation 0626
Social Work 0627
Sociology 0938
General 0631
Criminology and Penology 0628
Demography 0629
Ethnic and Racial Studies 0630
Individual and Family Studies 0700
Industrial and Labor Relations 0344
Public and Social Welfare 0709
Social Structure and Development 0999
Theory and Methods 0453
Transportation 0999
Urban and Regional Planning 0453
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture 0473
General 0285
Agronomy 0475
Animal Culture and Nutrition 0476
Animal Pathology 0359
Food Science and Technology 0478
Forestry and Wildlife 0479
Plant Culture 0480
Plant Pathology 0817
Plant Physiology 0777
Range Management 0746
Wood Technology 0306
Biology 0287
General 0308
Anatomy 0309
Biostatistics 0379
Botany 0329
Cell 0353
Ecology 0369
Entomology 0793
Genetics 0410
Limnology 0307
Microbiology 0317
Molecular 0416
Neuroscience 0433
Oceanography 0821
Physiology 0778
Radiation 0472
Veterinary Science 0786
Zoology 0760
Biophysics 0425
General 0996
Medical 0996

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences 0566
General 0300
Audiology 0992
Chemotherapy 0567
Dentistry 0350
Education 0769
Hospital Management 0758
Human Development 0982
Immunology 0564
Medicine and Surgery 0347
Mental Health 0569
Nursing 0570
Nutrition 0380
Obstetrics and Gynecology 0354
Occupational Health and Therapy 0381
Ophthalmology 0571
Pathology 0419
Pharmacology 0572
Pharmacy 0382
Physical Therapy 0573
Public Health 0574
Radiology 0575
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences 0485
Chemistry 0749
General 0486
Agricultural 0487
Analytical 0488
Biochemistry 0738
Inorganic 0490
Nuclear 0491
Organic 0494
Pharmaceutical 0495
Physical 0754
Polymer 0405
Radiation 0605
Mathematics 0986
Physics 0606
General 0608
Acoustics 0748
Astronomy and Astrophysics 0607
Atmospheric Science 0798
Atomic 0759
Electronics and Electricity 0609
Elementary Particles and High Energy 0610
Fluid and Plasma 0752
Molecular 0756
Nuclear 0611
Optics 0463
Radiation 0346
Solid State 0984
Statistics 0346
Applied Sciences 0984
Applied Mechanics 0984
Computer Science 0984

Engineering 0537
General 0538
Aerospace 0539
Agricultural 0540
Automotive 0541
Biomedical 0542
Chemical 0543
Civil 0544
Electronics and Electrical 0348
Heat and Thermodynamics 0545
Hydraulic 0546
Industrial 0547
Marine 0794
Materials Science 0548
Mechanical 0743
Metallurgy 0551
Mining 0552
Nuclear 0549
Packaging 0765
Petroleum 0554
Sanitary and Municipal 0790
System Science 0428
Geotechnology 0796
Operations Research 0795
Plastics Technology 0994
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,
Bulk Synchronous Parallel:
Practical Experience with a Model for Parallel Computing
submitted by
Anton Saarimaki
in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

April 1995

Abstract

Valiant proposed the Bulk Synchronous Parallel (BSP) as a possible model for parallel computing. He calls BSP a "bridging model" because it is applicable to both system and algorithm design. The model allows hardware and software design to proceed independently but ensures compatibility between computers and programs.

This thesis explores the practical use of BSP focusing on the portability and predictability offered, without any significant loss in efficiency. A BSP algorithm for sorting proposed by Gerbessiotis and Valiant is implemented in a portable fashion on three different parallel computers, specifically the Intel iPSC/860, the Transtech Parastation and the Alex AVX2. The program uses a standard library of functions designed and implemented for each machine to support the BSP model. The measured performance of the program is compared to the BSP predictions and other sorting results on similar machines to provide evidence for the utility of the BSP model.

Acknowledgments

Erika Dolphin, for balancing my life.

Peter Choynowski and Cheryl Schramm, for keeping the machines running and trying to answer all my questions.

Frank Dehne and Jörg Sack, for initiating and supporting my work on parallel computers.

Danny Krizanc, my very patient advisor, for debugging my English.

Access was provided to the Transtech Parastation computer through Murray Woodside of the Department of Systems and Computer Engineering, Carleton University.

Access was provided to the Alex AVX2 computer through an research and development agreement between Carleton University and ALMERC0 Inc.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	viii
List Of Figures	ix
1 Introduction	1
2 The Model	7
2.1 Bulk Synchronous Parallel (BSP)	7
2.2 Comments on BSP	10
2.3 Other Models	12
2.3.1 PRAM	12
2.3.2 LogP	12
2.3.3 C^3	14
2.3.4 Message Passing Libraries	14
2.3.5 Other BSP implementations	15

3	The Algorithm	17
3.1	Notes about the algorithm	19
3.2	BSP Analysis of the Algorithm	21
3.3	Implementation	22
4	The Machines	25
4.1	Intel iPSC/860	27
4.1.1	Implementation Details	28
4.1.2	BSP Parameters	29
4.2	Transtech Parastation	31
4.2.1	Implementation Details	32
4.2.2	BSP Parameters	34
4.3	Alex AVX Series 2	36
4.3.1	Implementation Details	37
4.3.2	BSP Parameters	39
4.4	Comparing and Contrasting the Machines	42
5	The Results	45
5.1	Portability	45
5.2	Predications	48
5.3	Efficiency	52
6	Conclusions	55
A	Graphs of Results	58
A.1	Intel Results	58
A.2	Transtech Results	63
A.3	Alex Results	69

B Common Source Code	77
B.1 BSP Sort Functions	77
B.2 Other Functions	84
C Intel Communication Library Source Code	88
D Transtech Communication Library Source Code	95
E Alex Communication Library Source Code	114
Bibliography	126

List of Tables

4.1	Intel BSP Parameters	29
4.2	Transtech BSP Parameters	34
4.3	Alex BSP Parameters	42
5.1	Speedup Shown by BSP Sort on All Machines	51
5.2	Intel Sort Comparisons	52
5.3	Transtech Sort Comparisons	52
5.4	Alex Sort Comparisons	53

List of Figures

1.1	Relation of Model to Algorithm Design, System Design and Programs	4
3.1	Pseudo C code for BSP sort	20
4.1	Intel <i>h</i> -relation Performance	30
4.2	Transtech <i>h</i> -relation Performance	35
4.3	Alex <i>h</i> -relation Performance	41
A.1	Intel Sort Times, 1 processor	59
A.2	Intel Sort Times, 2 processors	60
A.3	Intel Sort Times, 4 processors	61
A.4	Intel Sort Times, 8 processors	62
A.5	Transtech Sort Times, 1 processor	64
A.6	Transtech Sort Times, 4 processors	65
A.7	Transtech Sort Times, 8 processors	66
A.8	Transtech Sort Times, 12 processors	67
A.9	Transtech Sort Times, 16 processors	68
A.10	Alex Sort Times, 1 processor	70
A.11	Alex Sort Times, 4 processors	71
A.12	Alex Sort Times, 9 processors	72
A.13	Alex Sort Times, 16 processors	73
A.14	Alex Sort Times, 25 processors	74

A.15 Alex Sort Times, 36 processors	75
A.16 Alex Sort Times, 49 processors	76

Chapter 1

Introduction

Parallel computing hasn't enjoyed the same amount of success sequential computing has had. Several issues contribute to the difficulties parallel computing is experiencing. Many different architectures are used, ranging from fine grained massively parallel machines to workstations connected by an ethernet network. Architectures are changing rapidly as technology is rapidly evolving and new hardware becomes available. Many different programming models are used, for example, message passing or shared memory approaches. Models for parallel computing are either dependent on a particular architecture or not applicable to any machine. No standards are appearing, as the different models work well on the architectures they match or are used for theoretical analysis only. Algorithms are designed for a particular machine and often are not portable to other architectures or even to different configurations of the same machine. Programs must be rewritten to run on other machines or when upgrades occur. When portability is possible, there are often severe performance penalties. In general, parallel algorithms and programs are not portable and performance is not predictable.

On the other hand, sequential computing has been very successful. The workstation market is very large and is the driving force behind many of the advances in, and the rapid evolution of, technology. Programs are readily portable between

sequential computers. Program performance can be accurately predicted depending only on a few parameters that characterize the machine. The reason for this success is largely based on the common model that underlies all aspects of sequential computing. The von Neumann model was proposed over 50 years ago in "First Draft of a Report on the EDVAC" by J. von Neumann and in "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument" by A.W. Burks, H.H. Goldstine and J. von Neumann [56, 24]. In these papers and subsequent ones they presented a complete design for the EDVAC computer. The main contributors to the design were, the above authors, on the theoretical side and J.P. Eckert and J.W. Mauchly on the technical side, all working at the Moore School of Electrical Engineering at the University of Pennsylvania around 1945 [24]. The combination of theoreticians and hardware designers gave the model its strength.

The experience of building the ENIAC led to the von Neumann model. The ENIAC used too many tubes, was not easily adaptable to all problems, had a small memory, had a very long setup time and was physically very large. A quotation from [24], pages 204-205, shows the logic of the day and echoes the current situation.

... the delay line provided a means for storing a number at about 1/100 the cost in vacuum tubes of the ENIAC circuits. This was a first major step in bringing down the tube count. Another was to abandon the highly parallel operation of the ENIAC where many additions and a multiplication all proceeded in parallel. Thus, instead of 20 accumulators (they were of course adders) plus a multiplier and separate divide-square rooter, von Neumann's account envisages one arithmetic organ capable of just one operation at a time. The basic idea here was that the electronic technique is potentially so fast one can afford to abandon all parallelism and still obtain enough speed. This postulate has underlain virtually all machine developments ever since and is only in the last few years being questioned.

At the time when the von Neumann model was proposed, a "computer" was a person who computed. The model and the use of the word "organ" in its description

seems to reflect the human computer as the basis for the logical components. The von Neumann machine is made up of five "organs", the control organ, the arithmetic organ, the memory organ, the input organ and the output organ. The model sets out the logical relationships between the various components, leaving the details to the designers.

Various hardware solutions were suggested for the organs, but the technology was still evolving and being developed. The model had to be independent of any specific hardware solution and flexible enough to allow for any implementation. The model forms the basis for the computer system design which determines the hardware and operating system design. The final system must support the model.

They proposed an instruction set, implemented algorithms using it and analyzed their performance, all before the machine was built. To be sure that this work would be applicable to the final machine, the algorithms and the analysis were based on the model. Since the final system would support the model, programs based on these algorithms would be able to run, and the analysis would accurately predict performance. The model had to be robust enough to support different programming models and not restrict algorithm design.

Using the model, system design and algorithm design can proceed independently of each other. The dependencies on the same logical model, see Figure 1.1, ensure that programs based on the algorithms will run on the machines as predicted. Valiant calls this a "bridging" model [54] and it is the key characteristic behind the success of the von Neumann model.

A quotation from "The Preparation of Programs for an Electronic Digital Computer" by M.V. Wilkes, D.J. Wheeler, and S. Gill in [24], page 197, highlights the benefits the von Neumann model gave, resulting in smaller more powerful machines, with some measure of portability of personnel and algorithms, if not actual code.

Several machines working on the same principles as the EDSAC are now in operation in the United States and in England. These principles derive from a report drafted by J. von Neumann in 1946[sic]...It is found that machines designed along the lines laid down in this report

Figure 1.1: Relation of Model to Algorithm Design, System Design and Programs

are much smaller and simpler than the ENIAC and at the same time more powerful. The methods by which programs are prepared for all these machines are, as might be expected, similar, although details vary according to the different codes used. Anyone familiar with the use of one machine will have no difficulty in adapting himself to another.

The success of the von Neumann model and sequential computing suggest the characteristics a successful parallel computing model must have. It must be a “bridging” model, used by both hardware and algorithm designers. It must be implementable in existing (and future) hardware and be applicable to a wide range of machine architectures but it should not depend on any specific hardware configuration. Algorithms based on the model should be portable. It should direct the algorithm designers to efficient solutions. Algorithm analysis should be simple to perform, without requiring extensive or detailed knowledge of a particular machine. Analysis of an algorithm should provide accurate predictions of its performance on any machine.

Valiant, in [54], proposed the Bulk Synchronous Parallel (BSP) model as a possible bridging model for parallel computing. The BSP model satisfies the above requirements. It requires minimal system support while still being theoretically

robust.

A BSP computer consists of a set of components (von Neumann machines), each of which can send and receive messages from any other component (point-to-point communication), and facilities to synchronize the components. These requirements are satisfied by many different existing parallel machines and can continue to be supported by future hardware. Valiant discusses the possibility of supporting the BSP model using optical crossbars, for example.

The BSP model characterizes a machine using only three parameters, the number of components, which indicates the amount of parallelism available, the ratio of computation and communication speeds, which allows communication costs and computation costs to be compared and the time between synchronizations, which determines how coarse grained the algorithm must be. The parameters are usually measured for existing machines but can be deduced from analysis if required.

The model supports almost any programming model from message passing to real or simulated, shared memory. Performance predictions are based on amount of computation, the amount of communication and the BSP parameters. Valiant offers a proof [55] that the BSP model can optimally simulate the parallel random access machine, PRAM [19], with certain restrictions based on machine parameters and the amount of parallelism.

The simple abstraction of the communication system allows the model to be independent of any particular implementation and to be portable to any system. It also reflects Valiant's intended use of the model for predicting upper bounds for communication intensive algorithms, in which case, the overheads associated with communication are dominated by the time it takes to actually move the data. Algorithms that require little or no communication are handled well by other models.

This thesis explores the use of the BSP model of parallel computing during the design and implementation of a portable parallel algorithm. In particular, we focus on the ease of portable implementation and the accuracy of the performance predictions. Many papers dealing with parallel computing models only present theoretical results or focus on predictions made for one particular machine, even when

they are arguing for the practicality and portability of their approach. To overcome this problem, the same program was run on three different parallel computers by implementing a standard common library of functions that support the BSP model on each machine. To demonstrate the utility of the BSP model we compare actual runtimes to the BSP predictions and to results on similar machines.

Chapter 2 gives a detailed description of Valiant's BSP model, the requirements it makes of a parallel computer and the parameters it uses to characterize a parallel computer. Also included are responses to comments made about the BSP model and comments on some of the other parallel computing models. Chapter 3 presents a portable parallel algorithm for a basic computing problem, namely sorting. We analyze its performance using the BSP model and discuss some of the portable implementation issues. Chapter 4 describes the parallel computers that were used for testing, an Intel iPSC/860, a Transtech Parastation and an Alex AVX2. For each machine a short description of its hardware and communication system is given. Details about the implementation issues particular to each machine are discussed and the measured BSP parameters for each machine are listed. In Chapter 5 the results of implementing the algorithm in a portable fashion are discussed. The actual runtimes on the three computers (and their various configurations) are compared to the BSP predictions and to runtimes of other sorting programs on similar machines. Chapter 6 presents the conclusions reached about the use of the BSP model, focusing on the practical portability and predictability it offers without any loss in efficiency. The appendices contain the graphs of the actual runtimes versus the BSP predicted runtimes, the source code that is common to all of the machines and the source code that is specific to each machine.

Chapter 2

The Model

2.1 Bulk Synchronous Parallel (BSP)

The BSP model, as defined by Valiant in [53, 55, 54, 23] requires the following of a parallel computing system:

- processing and/or memory components;
- point-to-point communication between the components;
- facilities to synchronize all, or a subset, of the components.

For simplicity we assume each component has at least one processor with local memory and that there is a single user process per component. We use the word “process” and “processor” interchangeably to refer to these components. No extra information is required about the communication system, the interconnection network or the synchronization system.

The parallel system is characterized by the three parameters:

- p , the number of processors;
- g , a factor converting communication cost to computation cost;
- L , the minimum time between synchronizations.

A "superstep" is defined as the period between successive synchronizations. During a superstep a processor may perform three types of operations; work with data that was available in local memory at the start of the superstep, issue requests to send messages and issue requests to receive messages. When the communication actually takes place during a superstep is not known. The end of the superstep is marked by a synchronization event, during which the processors wait until all communication initiated during the superstep has been completed.

The amount of computation is measured as the number of basic operations the processor performs. The amount of communication is measured in words. It is assumed that the size of the data items involved in computation and communication are the same. The amount of communication in a superstep is the maximum of the number of words sent or the number of words received. Another option is to take the sum of the sends and receives. This would change the analysis by only a constant factor. The cost of communicating is determined by converting the amount of communication to an equivalent number of computation operations using g . The time to route an h -relation is less than or equal to the time to perform gh operations, for all h greater than some h_0 . During an h -relation each processor sends and receives at most h words, with each word being routed to a random destination. The factor g is the ratio of the computation bandwidth to the communication bandwidth of the system. The closer the value of g is to 1 the better the balance between the processor and the communication system.

L is measured in the equivalent number of operations that could be performed between two successive synchronizations. L is bounded below by the systems implementation of the synchronization function but the algorithm determines the upper bound. The larger L is, the coarser grained the algorithm must be, to offset the cost of synchronizing the processors.

The BSP model assumes that these parameters are available to the user. The compiler can make use of the BSP parameters to generate executables that operate efficiently on different machines. The algorithm can also make explicit use of the BSP parameters and the problem size to adjust its behavior at runtime.

Each superstep is independent of other supersteps because of the synchronizations that occur. The cost of an algorithm is the sum of the costs of its supersteps. The cost of a superstep, is determined by the amount of computation, C_P , and communication, C_C , that is done during the superstep. It is at least L , the time between synchronizations. There are several ways to combine these three factors to arrive at a cost for a superstep, for example $\max(C_P, C_C, L)$, $\max(C_P + C_C, L)$ or $(C_P + C_C + L)$. The choice isn't critical, producing results that differ by only small constant factors, but it should correspond to the behavior of the system and be convenient to use. We use the sum of the three factors, $(C_P + C_C + L)$.

The BSP model strives for simplicity and ease of use. It is a minimal specification that is still theoretically robust. Snir, in [48], argues that the fewer the parameters the easier it is to port codes from one machine to another. By abstracting the communication system and focusing on measurable attributes, BSP allows a great many factors to be considered without requiring detailed knowledge of the system. An analysis of the communication system performance of a machine based on its architecture, network topology, routing scheme, congestion, buffering, communication patterns, etc., soon becomes intractable, if not impossible.

The basic nature of the requirements BSP makes of a system allows the model to be applied to many different parallel computer architectures. It does not depend on any particular communication system implementation or network topology. It is capable of supporting many different programming models. It can be applied at different levels within an algorithm depending on the analysis required. We apply the model at the level of global communication functions. The communication systems of the machines we tested are slow enough that we must take full advantage of machine dependencies to try to increase communication speeds. For faster machines a lower level, Valiant calls "direct mode", can be used. At this level the user has complete control of the memory system and knows which references are local and which are remote. Explicit functions such as "get" and "put" are used to access data held by other processors. As machine performance improves, Valiant suggests an "automatic mode" system would be possible. The machine would control all

memory accesses and support a single address space. The user would not know if references were local or remote.

2.2 Comments on BSP

BSP is just one of many models that are in use or have been proposed. Some are more abstract and some more realistically model the hardware. BSP has been positioned somewhere between the two extremes so that it can be used by both system and algorithm designers. Each group with a model to promote criticizes the others. The following are the main points on which they base their comments.

The model is too abstract. By being very general BSP gives maximum freedom to the designers. The model is implementable with existing technology but the details are left to the system designer. Its simplicity allows the model to be supported by many different kinds of architectures. The model is also sufficient to support most programming models. This allows the algorithm designer to use the methodology that best suits the problem.

The model is not abstract enough. Skillicorn in [46, 47] argues for a more abstract model, the Bird-Meertens formalism, since BSP algorithms still depends on parameters that characterize a machine. This leads to having to alter, automatically or by hand, the algorithm for each new machine. Darlington and To [16] also argue for a more abstract model, namely skeletons, citing the low level and laborious task of using a conventional programming language as suggested by BSP.

The model omits locality. The omission is intentional. When locality is considered either portability is lost or the design of algorithms becomes prohibitively complex. The algorithm would first have to determine system topology and then decide how it must behave. Some models are themselves portable and consider locality but produce algorithms that are tailored to a particular machine and not portable. Basically, BSP considers only a two level memory, references are either local or remote but even this distinction is removed in automatic mode.

Reducing communication to a single factor g is unrealistic. BSP was designed to accommodate algorithms that make extensive use of the communication system, where the overheads of communication are dominated by the time it takes to actually move the data.

Communication patterns are useful only if the network topology is also known. By ignoring them BSP allows system designers to use the communication patterns that match the particular network they are working with. BSP can work with logical communication patterns, e.g., broadcast trees, and offer algorithms and predictions for such operations. However, the details are left to the implementor. In our case, the global communication functions take into account the network topology and communication patterns to try to improve performance.

By basing g on the most complex communication function, routing an h -relation, BSP overcharges other communication but maintains simplicity and upper bound predictions. Allowing g to be a function that depends on the amount of data being moved or having different g factors for different communication patterns would allow more accurate predictions to be made.

Requiring synchronization events is inefficient and overly restrictive. Synchronizations allow the system designer the freedom to schedule communication to best suit the actual network. Adjusting the size of the supersteps or partitioning the processors into groups that synchronize independently are methods for controlling the inefficiencies. Valiant mentions realizing synchronization in hardware in [54] but does not require it, noting there are many other ways to do it. Synchronizations and supersteps greatly simplify the design and analysis of algorithms and the extra cost is not prohibitive.

2.3 Other Models

2.3.1 PRAM

The PRAM [19] is very abstract and of theoretical interest only. The PRAM model forces synchronous SIMD algorithms that exploit unrealistic properties inherent in the model. Since real memory systems cannot be accessed in unit time, PRAMs cannot be built with existing hardware. The implementations that are currently being proposed and built [1] are actually simulations similar to that proposed by Valiant executed in hardware. Many have proposed extensions to the PRAM model to deal with these issues, including Valiant [55]. The precursor to BSP was called the XPRAM. Some of the aspects of the BSP model are related to XPRAM's original use as part of a proof of an optimal simulation of a PRAM.

Snir, in [48], argues that g will reach some constant value that scales with the number of processors, driven there by market forces. Others argue that if g ever reaches a constant value that scales with the number of processors, then the PRAM model can be used for programming.

2.3.2 LogP

LogP proposed by Culler *et al.* in [15] uses the BSP model as a starting point. They criticize BSP for being too abstract, so they tuned LogP to more accurately model a message passing system, specifically the CM-5, on which they implement examples to demonstrate the utility of their model. The model's complexity is increased with no real improvements in portability or analysis.

LogP is based on a distributed memory multicomputer, supporting point-to-point communication over an unknown network. It uses four parameters, p , the number of processor/memory modules, L , an upper bound on latency incurred in communicating a message from a source to its destination, g , the minimum time interval between consecutive sends or receives and o , the length of time a processor is busy with communication and cannot perform other operations.

The parameter L acts exactly like synchronization, guaranteeing message delivery within a set time period. The parameters g and o divide the measurement of the communication system into two parts, a bandwidth, that measures the performance of the network and an overhead, that measures the performance of the interface between the processor and the network. The network also has a fixed capacity, which no processor exceeds. At most L/g messages can be sent from a processor or to any one processor.

The model seeks only to expose some of the costs of parallel computing and makes no restriction on programming style. This gives the algorithm designer and programmer more freedom. By removing the requirement for synchronization events, there are some gains in performance, as processors no longer have to wait for others to complete a superstep, but the simplicity of analysis supersteps offer is lost and some of the flexibility supersteps give the system designer is also lost.

The authors give the following argument regarding the problems of abstraction. By abstracting the internal structure of the network into a few performance parameters, the model cannot distinguish between "good" and "bad" communication patterns. Various network interconnection topologies are known to have specific contention free routing patterns. For every network there are certain communication patterns where performance will be degraded due to internal contention among routing paths. These patterns are highly specific, often depending on the routing algorithm, and amount of buffering in each router, as well as the interconnection topology. This is a problem that is beyond the scope of LogP and BSP. Another model is required to provide a more detailed view of the communication system. (See C^3 below.) The authors mention that a possible way of extending the LogP model to reflect network performance on various communication patterns would be to provide a g value for each communication pattern. This could also be considered for BSP. In [57] it is shown that LogP can support an optimal PRAM simulation. Overall, the differences between BSP and LogP are quite small.

2.3.3 C^3

C^3 was proposed by Hambruch and Khokhar in [25]. The three C's refer to the three main factors that are considered by the model, Computation, Communication and Congestion. The model uses supersteps like BSP but the analysis of communication depends on detailed knowledge of the network topology, routing protocol, packet size, etc. This methodology is too complex, unless one were implementing a communication system. The model is portable and can be applied to many different kinds of machines but algorithms are then designed for one particular system and are not portable.

The authors provide theoretical analysis for most of the basic global communication patterns such as one-to-one, all-to-all, all-to-one, one-to-all, broadcast and reduction. Initial validation of the C^3 model is made through implementations on the Intel Touchstone Delta, but they give implementation results for only the one-to-all and all-to-one functions which don't suffer from congestion.

2.3.4 Message Passing Libraries

MPI [20] is the current popular standard for message passing. Based on contributions from many other systems, its aim is to provide an architecture and system independent interface for the programmer. Designed to run on heterogeneous systems, providing support for point-to-point communication functions (6 send, 2 receive and 2 send/receive operations), groups, contexts, communicators, message matching, datatypes, topologies, collective communication functions, global computation functions and many other features. It is a very large standard, defining 123 different functions. Current implementations of MPI usually use whatever native message passing system that is available, for example, NX/2 on Intel computers [33] or PVM [22] on networks of workstations. This means there are a lot of overheads within MPI and extra overheads due to the implementations on top of other systems.

MPI offers portability but no theoretical framework on which to base an analysis and no programming model to follow to ensure efficient implementations. A smaller

more concise model would greatly ease portability and focus attention on the true bottlenecks to performance.

2.3.5 Other BSP implementations

The Oxford BSP Library [40, 41, 10, 39] is a portable implementation of the BSP model. It provides only six basic operations, `bsp-start` and `bsp-finish` to begin and end parallel execution, `bsp-fetch` and `bsp-store` to transfer data, `bsp-sstep` and `bsp-sstep-end` to delimit supersteps. The authors have followed the fine grained aspects of BSP and produced a very low level implementation. It requires support for shared memory or systems with very fast communication networks to achieve reasonable performance. It also requires the user to follow a strict set of rules to manage parallel memory accesses. They provide some global communication functions that depend on special communication patterns, e.g., broadcast and reduce. BSP parameters for several machines based on the Oxford BSP library are given by Miller in [40].

In [13] Cheatham *et al.* propose, H-BSP, a general purpose parallel computing environment based on BSP. H-BSP consists of, BSP-L, an experimental high level programming language, a collection of compiler tools, which explicitly use the BSP parameters to generate efficient executables for a wide range of parallel computers and a library of communication and synchronization functions. For values of g close to 1, a simulated shared memory model can be efficiently supported. For larger values of g explicit use of the BSP parameters by the programmer or the compiler would likely yield better performance. The BSP parameters are explicitly used in the design of algorithms and they should also be present in the system implementation, compiler and operating system. Their approach is at a higher level than the Oxford BSP library.

There have been other implementations of BSP usually focusing on low level compliance with the BSP model in an effort to simulate PRAMs. In [45] BSP is implemented on a Meiko Computing Surface and later in [38] ported to use CHIMP [2] on the same machine.

In [35] a study is made of the routing required by BSP implemented on a network of transputers using a method developed by Valiant [52]. In [26] operating system features that would be necessary to support BSP and a higher level programming interface based on sequentially consistent shared memory are discussed.

Chapter 3

The Algorithm

To test the portability and predictability of the BSP model we chose to implement a parallel sorting algorithm. Sorting has been analyzed a great deal and many sorting algorithms have been proposed and tested on various parallel machines [11, 28, 49, 37, 36, 51]. Sorting is reasonably balanced with regard to the amount of computation and communication that needs to be performed. Sorting is also a basic operation that is often required as part of many other algorithms [17, 43, 30]. These aspects of sorting make it a reasonable problem to use to test the BSP model.

In [36], the authors classify parallel sorting algorithms according to two criteria; whether the sort is based on a merge function or a partition function and whether it routes data to its final destination in a single step or in a series of steps. BSP does not consider locality, so the distance a piece of data moves does not effect the analysis, only the number of times it is moved. Multistep routing is ruled out by BSP since the data is moved through the network more than once. The differences between merge or partition based functions is less clear and depend on the implementations involved.

The algorithm presented here is based on the sort presented in [23] which includes a more detailed BSP analysis. It is based on ideas from Quicksort [29], Samplesort [21], Flashsort [44] and sampling in a parallel system [31].

The parallel sorting algorithm, called BSP sort in this thesis, can be simply outlined as follows.

1. Randomly sample local data
2. Collect samples (all-to-one)
3. Sort samples and select pivots
4. Distribute pivots (broadcast)
5. Partition local data according to pivots
6. Route partitions (all-to-all)
7. Sort local data

The usual assumptions apply to BSP sort. At the start, the data is reasonably distributed among the processors participating in the sort. The data does not have to be perfectly balanced and will not be perfectly balanced after the sort has completed. An extra balance function could be performed after the sort, if perfect balance is required. Bad initial distributions should be improved by the sort. If the data contains many duplicate keys, the balance will be thrown off. An analysis of this problem for a merge based parallel sort is presented in [37].

The algorithm consists of 7 supersteps. Supersteps 1, 3, 5 and 7 perform only computation and supersteps 2, 4 and 6 perform only communication. During the first superstep, all the processors randomly sample their local data. In the second superstep, all the processors send their samples to one processor. This operation is generally called an all-to-one communication function. In the third, that one processor sorts all the samples and selects the data values, pivots, to be used to partition the data among the processors. In the fourth, the one processor distributes the values of the pivots to all the other processors. This operation is generally called a broadcast communication function. In the fifth, all the processors partition their local data according to the pivot values. In the sixth, all the processors route the partitions to their final destinations. The first partition on each processor is sent to the first processor, the second partition on each processor is sent to the second processor and so on. This operation is generally called an all-to-all communication function. In the seventh and last superstep, all of the processors sort the data they

have stored locally, consisting of all of the partitions just received from all of the other processors.

The algorithm as presented, leads to a simple, straightforward implementation. A slightly simplified version, datatype details removed, of the C source code is shown in Figure 3.1. The actual C source code can be found in Appendix B.

3.1 Notes about the algorithm

The original sorting algorithm by Gerbessiotis and Valiant was recursive but they argued that usually only one or two passes were expected in practical situations. On the size of machines we are considering recursion is not necessary. Following the BSP analysis exactly, recursion is not suggested, since it involves communicating the data more than once. Only if one considers locality and message routing, both excluded from BSP, does a recursive algorithm have some merit.

The original version also required the sample to be taken randomly from the entire data set. We have each processor randomly sample its local data only. This method was analyzed in [11] and it was proved that the local choice method is at least as good as the global choice method. In particular, they prove, that given, p , the number of processors, n , the total number of data items, n_{max} , the maximum number of data items a processor receives and m , the local sample size, the probability that any one processor receives more than $\alpha n/p$ is:

$$Prob[n_{max} > \alpha n/p] \leq ne^{-(1-1/\alpha)^2 \alpha m/2} \quad (3.1)$$

Given a large n and a sample size around $\sqrt{n}$, the probability that a processor receives more than 1.5 times the expected number of items is extremely small.

The algorithm does not make any explicit use of the BSP parameters. When it runs on machines with different BSP parameters its behavior will not change but its runtime might. No details are given about the communication functions so they could make use of the BSP parameters depending on the implementation.

```
/* p    the number of processors being used */
/* me    the unique processor identifier (0, p-1) */

parallel_sort(data, n, max_n, sample, m, max_m) {
    /* data, array of data items to be sorted */
    /* n, number of data items */
    /* max_n, size of data array, max_n > n */
    /* sample, array for data samples */
    /* m, number of samples to take locally */
    /* max_m, size of sample array */

    if (p > 1) {
        for (i=0; i<m; i++) sample[i] = data[rand()%n];
        barrier();
        m = all_to_one(0, sample, m, max_m);
        barrier();
        if (me == 0) {
            local_sort_many(sample, 0, m-1);
            step = m / p;
            for (i=0; i<p-1; i++) pivots[i] = sample[(i+1)*step];
        }
        barrier();
        bytes = broadcast(0, pivots, p-1);
        barrier();
        multiple_partition(data, n, pivots, sizes);
        barrier();
        n = all_to_all(data, n, max_n, sizes);
        barrier();
    }
    local_sort_many(data, 0, n-1);
    barrier();
    return(n);
} /* end of parallel_sort */
```

Figure 3.1: Pseudo C code for BSP sort

3.2 BSP Analysis of the Algorithm

Each of the supersteps is implemented as a short sequence of statements or a function call with explicit barriers occurring between them. Without these barriers the algorithm has logical synchronizations during the global communication functions but some overlap of the supersteps could occur and this could slightly improve performance.

For the following analysis, assume p is the number of processors, g is the ratio of computation and communication speeds, m is the local sample size (pm is the total sample size) and n is the local number of data items (pn is the total number of data items). An extra space factor ϵ must also be considered because the data partitioning scheme does not guarantee perfect balance. $n(1 + \epsilon)$ is the maximum number of data items a processor could have after the all-to-all communication phase. The cost of the all-to-all and the final local sort are based on this number. The supersteps have the following costs:

1. m , locally pick m data items
2. gpm , all-to-one, route pm samples
3. $pm \log(pm)$, sort all of the samples
4. $g(p - 1)$, broadcast, route $p - 1$ pivots
5. $n \log(p - 1)$, partition data according to pivots
6. $gn(1 + \epsilon)$, all-to-all, route n data items
7. $n(1 + \epsilon) \log(n(1 + \epsilon))$, sort local data

For the machine and problem sizes tested, there are approximately tens of processors, samples sizes in the thousands and millions of data items. Considering this, it is easy to see that supersteps 6 and 7 should dominate the runtime.

The total time for the sort algorithm is given by the sum of the superstep times plus the cost of the seven synchronizations that occur.

$$\begin{aligned} \text{Total Cost} = & m + pm \log(pm) + n \log(p-1) + n(1+\epsilon) \log(n(1+\epsilon)) + \\ & g(pm + (p-1) + n(1+\epsilon)) + 7L \end{aligned} \quad (3.2)$$

The space required by the algorithm is the maximum of the data size plus the size of the expected message size in the all-to-all communication function times the extra space factor, $(n + n/p)(1 + \epsilon)$ or the data size plus the total sample size, $(n + pm)$. Sample sizes are usually about $\sqrt{n}$. We attempt to minimize the unbalance, represented by ϵ , that is caused by sampling to determine the partitions. This reduces the space requirements of the algorithm and improves runtimes. For most n we use $\epsilon = 0.5$, and this value is used for all of the predictions. For larger n , when memory limits must be considered, empirical results indicate $\epsilon < 0.1$ is possible.

3.3 Implementation

The sequential C source code for the computation only supersteps is portable once some system dependent functions such as, timing functions and error handling, are taken care of. Supersteps 2, 4 and 6 define the basic global communication functions that are required for the sorting algorithm, all-to-one, broadcast and all-to-all. Also required is a function to synchronize all of the processors, namely barrier. These 4 functions make up the communication library and once it is implemented on a particular machine, the sorting algorithm will be able to run on it.

A common set or library of communication routines is becoming a popular idea. Each manufacturer provides their own communication functions, e.g., Intel's NX/2 and the INMOS's transputer operating system. Common libraries were developed that specified a standard set of functions that could be implemented on a whole range of machines, e.g., PVM and Trollius, to provide some measure of portability. Now these many libraries and systems are becoming standardized. Through the efforts

of the Message Passing Interface Forum a standard, if large, set of communication functions have been defined [20]. Many different MPI implementations now exist for many different machines, based on each machine's native message passing system.

Valiant [54] argues that the basis for the communication system be an efficient implementation for routing arbitrary h -relations. The all-to-all communication pattern arises often in many important algorithms and clearly is the densest form of communication possible, stretching the communication capabilities of the interconnection network to the limit. We use the all-to-all communication function to implement h -relations. We also restrict the amount of space the all-to-all function uses. Its implementation therefore requires special consideration.

In [42], the authors discuss complete exchange or all-to-all personalized communication algorithms. They start with general algorithms then adapt them to suit the CM-5 architecture to improve performance. Our implementation of all-to-all is similar to their PEX algorithm. Instead of pairwise exchanges, in each round a processor sends a message to the processor with id equal to the local id plus an offset, modulo the number of processors and it receives a message from the processor with id equal to the local id less the offset, modulo the number of processors. It is a simple arrangement, not dependent on the network topology and accommodates configurations that are not powers of 2. A single buffer is used for both sending and receiving messages. As data is sent, more room is created for incoming messages. Only enough space to buffer one message is required at each processor because of the matching of the sends and receives. When handling large messages the communication required for the all-to-all can easily overwhelm the communication system, so the all-to-all implementations try to consider the network capacity and do some flow control.

The parameters required by each of the 4 functions that make up the global communication library are described next. The barrier function requires one parameter, the id of the processor that will coordinate the synchronization. For the communication functions, all sizes are in bytes, the datatypes of the messages are not considered. Each function returns the size of the message in the buffer after com-

munication is done. The receiver is responsible for allocating a buffer large enough to accommodate the incoming data. The all-to-one requires 4 parameters, the id of the root processor that will receive all the messages, a pointer to the message buffer, the size of the message and the size of the message buffer. The broadcast function requires 3 parameters, the id of the root processor that has the data to be distributed, a pointer to the message buffer and the size of the message buffer. The all-to-all function requires 4 parameters, a pointer to the message buffer, the total size of all the messages in the buffer, the size of the buffer and an array of the message size for each processor.

If portability is more important than performance the global communication functions can be implemented using the point-to-point communication functions required by BSP. If performance is more important than portability a machine dependent implementation can be used since the details of the global communication functions are not defined. In either case, it is not expected that the average user would implement the communication library. People familiar with a particular system would provide these functions, tuning them to take full advantage of functions supplied by the native system and information about communication patterns, network topology, routing, etc., to provide the maximum performance. Valiant provides a BSP analysis for broadcasting and parallel prefix function in [54, 23]. Oxford BSP supplies some high level functions that use special communication patterns and combining functions at the nodes. Various MPI implementations offer global communication functions implemented in terms of point-to-point functions but suggest using machine dependent functions to increase performance. For the machines studied here, network speeds are much slower than the processor speeds so the communication functions are machine dependent. As machines with faster networks are produced and more system support for communication is provided this layer of machine dependence can be lowered increasing portability without sacrificing communication speed.

Chapter 4

The Machines

The parallel sorting algorithm was implemented on three different computers, an Intel iPSC/860, a Transtech Parastation and an Alex AVX2. This chapter presents a description of each machine, the hardware and message passing systems, some details about the implementation of the communication library and the BSP parameters for each machine configuration.

The measurement of performance is poorly defined and comparisons must be made carefully [5, 14]. We assume that if the predictions of runtimes and the actual measurements of runtimes are made in some consistent manner they can be compared. The BSP parameters must also be measured in a consistent way, if their values and predictions based on these values are to be compared. This requires defining how the BSP parameters are to be measured.

The implementation of the sorting algorithm handles 4 byte floating point data-types. To find the computation bandwidth, g_P , loops consisting of basic operations, e.g., multiply and compare-swap, manipulating arrays of 4 byte floats were timed. The loop overheads were subtracted to arrive at an approximate operation rate measured in floating point operations per second (flops/s). Since we are considering sorting, the compare-swap rates were used to determine g_P .

To find the communication bandwidth, g_C , the performance of the all-to-all communication function routing arbitrary h -relations was measured. During an arbitrary h -relation each processor sends and receives exactly h words, each word

being routed to a random destination. The values of h divided by the time to route the h -relation determines performance measured in words per second (words/s). Words are 4 byte floats to match the computation bandwidth measurements. To calculate g_C , the rates for the h -relation were averaged for h large enough to produce messages with expected size greater than or equal to 1024 words. This corresponds to the internal buffer size on the Intel, 1/2 the packet size on the Transtech and the packet size on the Alex. For each of the machines this closely corresponds to the level portion of the h -relation performance graphs.

The value of g is the ratio of the computation to communication speeds and has units floating point operations per word (flops/word). In homogeneous systems this is just the speed of a single processor compared to its communication bandwidth. In heterogeneous systems this can be generalized to the the total processing speed compared to the total communication bandwidth. Since g is based on measurements of the system, there is some choice in determining its value, depending on the kind of predictions that are required and the accuracy desired.

In many papers [9, 18, 8] the communication system's performance is measured only by the bandwidth of an echo function, where a message is sent from one processor to another and returned. The researchers test the functions included in the native communication system supplied by the manufacturer but in a very unrealistic situation, a single message traveling through the network with no other communication traffic. Most testers are looking to identify simple communication cost equations based on overheads and bandwidth without considering any other factors. Testing here shows that there is little relationship between the performance of a single point-to-point communication and the performance of a complicated, communication intensive function like an h -relation. Many different factors effect the h -relation performance, including congestion at nodes and links, routing, buffering, packetization, which are not easily analyzed. System designers usually only supply point-to-point communication functions and possibly functions that have special communication patterns, such as broadcast or parallel prefix. More complex functions, such as all-to-all, are usually not included, even though their complexity

requires detailed knowledge of the communication system and network to avoid high overheads and achieve reasonable performance without overwhelming the communication network.

The value of L is based on measurements of the barrier function. The time required to run the barrier function is measured in seconds, L_T , and then converted to a number of operations using gp .

All results in this paper are calculated using wall clock times of compiler optimized C code. They are much lower than the advertised performance figures for these machines, since no effort has been made to tune the code by hand. This realistically reflects the performance an average user would expect to achieve.

Appendix B lists all the C source code that is common to all of the machines tested. This includes all the non-communication functions required for the parallel sorting algorithm and a random number generator. The same random number generator was used on each of the machines, so the data streams being sorted were the same for each machine.

4.1 Intel iPSC/860

The Intel iPSC/860 [34] used for these experiments was acquired in 1991 and is housed in a large floor standing cabinet with display lights that indicate the activity of each of the processors. The machine consists of 8 nodes arranged in a fixed hypercube network topology. Each node is an Intel i860/RX processor running at 40 MHz with 16 MB of memory and an FPU. The *icc/NX Sun4 Rel 4.0* C compiler by Intel and Portland Group was used. Each node runs the *NX/2*, Network eXecutive, operating system which can support only one process on each processor. The *NX/2* communication library provides blocking and nonblocking point-to-point communication, synchronization, broadcast and some special global operations. Messages are matched by receiver, sender and a user defined type field. The machine has one internal hard drive which all the nodes can access. The Intel is accessed through an attached host, the SRM (System Resource Manager), which is connected to the

local ethernet network. Up to 8 users can node (or space) share the system, each allocating an independent set of nodes. Input and output from any node to the user's system is supported.

See [9, 6, 27, 18] for detailed studies of the computation and communication performance of the the Intel iPSC/860. Two points from [27] that are of interest.

First, each iPSC node consists of three main components: a processor, local memory and a DCM (Direct Connect Module). The DCM controls the channel used for communication with neighboring nodes. The DCM has two 4096 byte queues that buffer all processor communication, one for incoming messages and one for outgoing messages.

Second, a different protocol is used for sending messages depending on their size. Small messages, less than or equal to 100 bytes, are usually sent immediately because each node has a number of 100 bytes buffer available to receive messages from other nodes. For large messages, greater than 100 bytes, a query is first sent to the receiver to check that enough space is available to store the message.

The maximum single channel hardware data rate is 2.8 million bytes per second, or 700,000 words per second. The measured performance of the NX/2 send and receive functions handling a single large message, 4096 words, on an otherwise unloaded network, to an adjacent processor, 1 hop, is 673,000 words per second and to a processor, 3 hops away, is 667,000 words per second. This is close to the hardware limit and with shows little loss due to multiple hops.

4.1.1 Implementation Details

Appendix C lists the C source code for the communication library for the Intel iPSC/860.

The barrier function is implemented using the NX/2 synchronization function. The broadcast functions uses the broadcast capabilities of the native send function. The all-to-one function uses blocking communication functions. The all-to-all function uses nonblocking communication functions.

p	g_P (flops/s)	g_C (words/s)	$g = g_P/g_C$ (flops/word)	L_T (seconds)	$L = L_T * g_P$ (flops)
1	2213000	-	-	-	-
2	2213000	581000	3.81	0.00013	288
4	2213000	308000	7.19	0.00026	575
8	2213000	245000	9.03	0.00042	929

Table 4.1: Intel BSP Parameters

During the all-to-all function messages are sent and received inside a loop. A processor initiates a nonblocking send and a nonblocking receive operation. It waits for the send to complete and then waits for the receive to complete before proceeding. Destinations are calculated using the offset-modulo method discussed earlier. Other methods dependent on the hypercube topology did not greatly improve the performance of the all-to-all. The machine tested is very small so that many of the possible network dependent characteristics do not have any impact.

4.1.2 BSP Parameters

The measured performance of the i860 processor was 3,821,000 multiplications per second and 2,213,000 compare-swap operations per second.

In Figure 4.1 the results of the h -relation tests are shown. Performance flattens out for messages with expected size greater than 256 words. The large dips around messages with expected size of 25 words are due to the protocol change for messages larger than 100 bytes. Slight dips around messages with expected size 1024 words, reflect the size of the DCM communication buffers. The rates for small h values indicate the relatively low overheads involved in the communication system and the all-to-all implementation. The maximum h -relation rates are about 100 times the minimum rates.

The BSP parameters for the Intel can be found in Table 4.1. The values of g are not too large indicating a reasonable balance between computation and com-

Figure 4.1: Intel *h*-relation Performance

munication speeds. The NX/2 synchronization function must be very efficient since the L_T times are very low. The parameters scale about as one would expect for a hypercube network topology.

4.2 Transtech Parastation

The Transtech Parastation [50] was purchased in 1992 and is housed in small box that sits under the monitor of the workstation it is connected to. It consists of 16 Transputer T805 processors [32] arranged in a partially configurable network. Each T805 has 16 MB of local memory, an FPU and runs at 20Mhz. The INMOS Version 2.01.10 C compiler by Inmos was used. Each T805 has four links, two of which are fixed and two of which connect to a configurable switch. On the machine tested the configurable links ran at half the speed of the nonconfigurable links. The fixed links group the processors into pipelines of 4 processors each. Altering the configuration is rather difficult, sometimes involving changing jumpers on the hardware itself. By default the topology is a mesh and was not altered because of the difficulty of reconfiguration and to accommodate other users of the machine. The machine was used as a pipeline to minimize the number of slow links used and simplify the communication system. Only 1 node of each allocated set has input and output capabilities with the external system. The machine can be space shared by up to 4 users, each allocating an independent set of nodes. Access to the Transtech is through the workstation it is connected to and the workstation is accessible over the local ethernet network.

The transputer operating and communication system is implemented mostly in hardware and allows multiple processes per processor. We use only one user process per processor but the communication system runs two extra processes per link being used. Processes can run at high or low priority. When running at high priority, a process runs until it executes one of a small number of functions dealing with communication. It then blocks and is descheduled. When running at low priority, a process runs until it blocks for communication or a time limit is reached and then is

descheduled. Preference is given to high priority processes when scheduling which process will run next. We ran all of the processes at high priority, so care had to be taken that one process did not exclude all others from running.

Communication is based on channels and allows communication to other processes on the local processor and to processes on adjacent processors, connected by a link. Messages are matched by use of the same channel. Multiplexing messages on the same channel is possible but must be managed by the user. A simple protocol manages process communication. The first process to call a communication function on a particular channel blocks until another process calls the matching communication function on the same channel. The message is transferred and both processes are added to their respective ready to run queues. The behavior is identical for messages passed between two processes on the same transputer as for processes on adjacent processors, although the underlying hardware mechanisms are different.

4.2.1 Implementation Details

Appendix D lists the C source code for the Transtech Parastation, the basic point-to-point communication system and the global communication library.

The very basic nature of the communication functions offered by the transputer system required that a point-to-point communication system had to be implemented first. The pipeline configuration simplified this task. The system implemented supports packetization, routing, buffering, nonblocking communication, error handling and synchronization. Packets are 8192 bytes long. Each link process can independently store 3 packets. To prevent communication from causing the user process to block, a user send or receive operation just adds the request to a global queue, from which it will be handled by one of the link processes. Messages are matched by receiver only. In a BSP fashion, it is assumed there would be barrier synchronizations between communication operations. This assumption makes such a simple implementation feasible.

The send and receive functions are implemented as distinct processes on the transputer, one send process and one receive process for each link being used. The

operation of the processes are linked together by global semaphores and message queues. They were implemented in such a way that they could operate using the same user defined buffer. The receive processes handle message routing as well as messages destined for the local processor. To receive a message three parameters are required, a pointer to a buffer, the amount of space available in the buffer and the number of messages to receive. They are global variables and are available to both the user and receive processes. When the space available is not large enough to accommodate the incoming message, the receive process blocks. As the send operation, using the same buffer, proceeds, it signals the receive process that more space has become available. The communication functions aren't required to use the same buffer but doing so conserves space.

The maximum hardware single channel link rate is 20 million bits per second. Due to the transputer transfer protocol used, this is effectively 1,740,000 bytes per second or 435,000 words per second. The performance of the point-to-point communication system we implemented, for a single large message, 4096 words, in an otherwise unloaded network to an adjacent processor, 1 hop, is 424,500 words per second and to a processor 15 hops away, end to end on the 16 processor pipeline, is 44,000 words per second. The 1 hop rate is very close to the hardware rate. The communication system and the extra processes do not add much extra overhead. The 15 hop result is more than 80 per cent of the theoretical value based on the 1 hop rate and a packetized store and forward routing scheme. The extra loss is due to overheads incurred handling the message packets at each processor.

The global communication library was implemented on top of this point-to-point communication system. The barrier function uses the synchronization support provided by the receive process to wait for all of the processors to call the barrier function and the broadcast function from the communication library to signal all the processors to restart. The other functions are implemented using the nonblocking point-to-point communication functions.

To save space the all-to-all uses a single buffer and uses the offset-modulo method to determine destinations. The all-to-all function uses two different protocols de-

p	g_P (flops/s)	g_C (words/s)	$g = g_P/g_C$ (flops/word)	L_T (seconds)	$L = L_T * g_P$ (flops)
1	281000	-	-	-	-
4	281000	228300	1.23	0.0010	281
8	281000	91800	3.06	0.0017	478
12	281000	58000	4.84	0.0023	646
16	281000	40500	6.94	0.0030	843

Table 4.2: Transtech BSP Parameters

pending on the expected message size. If the messages are expected to be less than one packet, a single receive is initiated for all of the incoming messages. The sends to each processor proceed in turn inside a loop, signaling the receive process as they complete. If the messages are expected to be larger than one packet, the receive is included in the loop and expects only a single message. The send signals the receive process after each packet is sent. Both the send and the receive complete before the next iteration occurs. This method requires very little buffering at each of the nodes.

4.2.2 BSP Parameters

The measured performance of the T805 processor was 517,000 multiplications per second and 281,000 compare-swap operations per second.

The results of the h -relation tests can be found in Figure 4.2. The performance is relatively level for h greater than 2048 words. The rates for low h values give a measure of the overheads involved. The dips that occur as the expected message exceeds 2048 words are due to the overheads of packetization. The maximum h -relation rates are about 1000 times the minimum rates.

The BSP parameters in Table 4.2 show very good balance. The speed of the processor is relatively slow so it is easier for the communication system to match it. The assumptions made about the network topology and usage of the communi-

Transtech Parastation h-relation Performance

Figure 4.2: Transtech *h*-relation Performance

cation system allowed for a the simple communication system design that could be implemented close to the hardware and that avoids large overheads. This resulted in a fast communication system that achieves speeds close to the hardware limits.

The parameters scale as expected for a pipeline topology. Some of the drop off of speed for configurations larger than 4 processors are a results of the slower communication speeds of the configurable links. For instance, the 8 processor pipeline is made up of two 4 processor fixed link pipelines connected by a slow configurable link, similarly the 12 processor pipeline has 2 slow links and the 16 processor pipeline has 3.

4.3 Alex AVX Series 2

The Alex AVX Series 2 [4] was purchased in 1993 and is housed in large floor standing cabinet. The system tested consisted of 62 compute nodes and 2 video nodes. The video nodes consist of a Transputer T805 and hardware support for video display monitors. They were not used during these tests. Each compute node consists of a Transputer T805 [32] with 8 MB of private memory and a FPU, running at 20, 25 or 30 MHz and an Intel i860XR [33], (no mention of a FPU), with 32 MB of memory shared with the T805 running at 40 MHz. The TCX Compiler V91.1 C compiler by Quality Computer Systems and Logical Systems was used for the T805 and the PGC/i860 Sun4 Rel 2.2 C compiler by Portland Group was used for the i860.

The T805 can run multiple processes. The operating and communication system used on the Alex is called Trollius [3, 8] and runs as a set of 18 processes on the T805. The system tested was running Alex-Trollius 2.2.2.1d, compiled locally to include local customizations and bug fixes. The i860 can run only one process and basically runs as a co-processor to the T805. It does not run Trollius and communication or timing requests must be handled by the T805. Each T805 has 4 communication links which are all configurable. The system requires one T805 to use one link for the connection to the external system. This breaks the symmetry of some network configurations. A complete torus topology is possible only if an extra node is used.

The configuration is determined by the user when allocating a set of nodes. For testing, the system was configured as a square mesh, varying from 1 by 1 up to 7 by 7. Up to 8 users can node (or space) share the system, allocating independent sets of nodes. Access to the system is through workstations that are connected directly to the Alex machine.

Trollius supports blocking point-to-point communication multicasting, multireeling and some other special global functions, e.g., parallel prefix. It also supports IO from any processor. There are functions that can check the status of the processes, both user and Trollius. Messages are matched by the receiver's node id, event field and a bitwise comparison of the type field. The T805 and the i860 have the same logical node id so the user must ensure the message reaches the correct processor by using the event and type fields.

There are four levels of message passing functions: kernel, data link, network and transport. Each level is built on the previous level and offers more functionality but at the cost of larger overheads. Kernel layer functions directly access the transputer hardware. Data link functions manage communication to adjacent processors. The network layer offers point-to-point communication, routing and packetizing of large messages. Packets are 4096 bytes long. If different processors send messages that consist of more than one packet with the same event and type to one receiver, the packets can become mixed because the senders id is not considered during message matching. The transport layer offers flow control and enables multiple processes to send multi-packet messages to a receiver using the same event while maintaining the atomicity of the messages. The sender and the receiver exchange messages agreeing on a new event value before the data transfer begins. Trollius uses a fixed routing scheme that is determined when nodes are allocated.

4.3.1 Implementation Details

Appendix E lists the C source code for the communication library for the Alex AVX2.

The barrier function is implemented using the Trollius multireel and multicast facilities. The broadcast function also uses multicasting. The all-to-one function is based on the Trollius network layer point-to-point communication functions but manages the packetization of messages to prevent packet mixing. The all-to-all function is implemented using Trollius transport layer functions.

For testing, the ordering and destination of the sends and receives in the all-to-all function was determined by an input file. This allowed us to experiment with different arrangements and to try to control the load on the communication system for large meshes. The same modulo-offset method to control the all-to-all was used. More elaborate pairing methods did not lead to any significant improvement in performance.

Two possible reasons for this are: the routing of messages is fixed for a given topology and the routes taken by messages are not always shortest path. When nodes are allocated, the host computer calculates the routing tables for each node and passes them into the node. A function, called map, calculates the routing table for each node. The order in which the nodes are handled is random. The function tries to control the congestion on each link by adjusting the weights associated with each link when it is used as part of a routing path. The first node that has its routing table calculated will get good, shortest path routes. The last node to have its routing table calculated might not get shortest path routes as map has to deal with all the results of all of the previous nodes routes. There is some tuning of the paths that can be done, by specifying the weights to be considered, the order to add paths and by explicitly defining paths. The average user should not be expected to work at such a low level.

The configurable nature of the Alex communication network makes it desirable to have communication functions that don't depend on a particular topology. Initially it was assumed that Trollius point-to-point communication system would be able to support topology independent global communication functions with little extra control or support by the implementors. This turned out not to be the case, as Trollius is very sensitive to communication problems as pointed out in [3].

In the original scheme half of the processors are sending and half are receiving at any one time. This effectively reduces by half the potential bandwidth of the network but lessens the buffering required. This scheme overloaded the communication system for large meshes. The problems stem from cyclic dependencies that can arise between messages. Trollius' fixed routing scheme and the inability of messages to back off and release network resources can cause the communication system to fail. The first attempt to solve this problem was a reduction in the number of active senders but at a corresponding reduction in performance. When run with a single active sender the all-to-all is guaranteed not to fail but is extremely slow. For large meshes it appears that locality and network topology must be considered.

The type of failures experienced here, in the point-to-point communication system of Trollius, correspond to the problems with abstraction cited by Culler *et al.* in their discussion of the LogP model [15]. By abstracting away network details the model cannot distinguish between "good" communication patterns and "bad" communication patterns. An analytical analysis of the Trollius system would not be able to predict the poor performance of the communication system. The problems lie deep inside Trollius at its lowest layers. The communication system performance suggest, that only algorithms which have a very high computation cost as compared to communication cost should be considered. In an effort to improve communication performance, Trollius is being tuned for the Alex AVX2 architecture and different algorithms for the all-to-all communication function are being investigated.

4.3.2 BSP Parameters

The measured performance of the i860 processor was 1,087,000 multiplications per seconds and 1,895,000 compare-swap operations per second. The Intel i860 multiply rates are much faster so the Alex i860 processors probably do not have FPUs. The measured performance of the T805 processor was 421,000 multiplications per second and 233,000 compare-swap operations per second. The Transtech T805 operation rates are slightly higher so the Alex T805s are probably running at 20 Mhz also.

The T805 performance in an active system would in fact be much less since it would be running the Trollius processes as well. Any communication that the T805 has to handle would take processing time away from the user process. It is assumed that the user process would run on the i860 leaving the T805 to run Trollius and handle communication.

The maximum hardware single channel link rate is 20 million bits per second. Due to the transputer hardware transfer protocol used, this is effectively 1,740,000 bytes per second or 435,000 words per second. Performance of the Alex-Trollius network layer communication functions for a large message, 4096 words, in an otherwise unloaded system, to an adjacent processor, 1 hop, is 60,400 words per second and to an a processor 12 hops away, corner to corner on a 7 by 7 mesh, is 19,400 words per second. The 1 hop rate shows the very high overheads encountered using Trollius. Each layer adding another extra penalty. The 12 hop rate is slightly higher than predicted based on the 1 hop rate and a packetized store and forward routing scheme. The reason for this is probably the way Trollius buffers messages, allowing the packets of a message to catch up to each other.

The results of the h -relation test can be found in Figure 4.3. The performance flattens out for messages with expected sizes greater than 2048 words. The very low rates for $h = 1$ indicate the very high overheads involved with Trollius. The large dips for large messages on large meshes are the result of the change in protocol necessary to avoid overloading the communication system. The maximum h -relation rates are about 10,000 times the minimum rates.

The BSP parameters for the Alex can be found in Table 4.3. The g values are very large and not close enough to 1 to make a shared memory simulation a viable option. The parameters don't scale as expected for a mesh topology. The combination of the fast i860 chip, the slower transputer communication speeds and the overheads from Trollius result in very poor BSP parameter values. Computation based on the i860 processor is much faster than communication based on the T805 running Trollius. Even if the maximum communication rate the T805 hardware provides could be used, when compared to the computation speed of the i860 it

Figure 4.3: Alex *h*-relation Performance

P	g_P (flops/s)	g_C (words/s)	$g = g_P/g_C$ (flops/word)	L_T (seconds)	$L = L_T * g_P$ (flops)
1	1895000	-	-	-	-
4	1895000	57300	33.07	0.0129	24446
9	1895000	30000	63.17	0.0168	31836
16	1895000	21700	87.33	0.0182	34489
25	1895000	18100	104.70	0.0208	39416
36	1895000	11900	159.24	0.0232	43964
49	1895000	7600	249.34	0.0262	49649

Table 4.3: Alex BSP Parameters

results in a g value of 4.4. The extra overheads incurred using Trollius result in a very slow communication system. The values of L are also very large because the synchronization function is implemented using the communication system. Based on the BSP parameters in Table 4.3, one might argue that the balance between computation and communication would be much better if the user process ran on the T805. If runtime is the main criteria this is probably not a good idea.

4.4 Comparing and Contrasting the Machines

The three machines studied have many similarities. They are all coarse grained distributed memory parallel computers. They all use off the shelf, commodity, chips taking advantage of advances in sequential processing technology. The machines have a relatively small number of processors with local memory connected by a communication network. The processors communicate with each other by explicitly sending and receiving messages over the network. The same processors that are used in the Intel and Transtech machines are used in the Alex. An i860 processor is used for computation in both the Intel and the Alex. The communication system runs on T805 processors in both the Transtech and the Alex. However, in the Intel the

i860 is also handling communication and in the Transtech the T805 is also doing computation. Each of the machines supports the C programming language. Each machine allows multiple users to space share the machine.

From a high level view the machines share many similar features but there are two high level differences that are very important: the network topologies and the communication system support. The Intel has a fixed hypercube topology. The Transtech has a partially configurable network, used as a pipeline for testing. The Alex has a completely configurable network, used as a mesh for testing. If the algorithm is dependent on a specific topology, this difference alone could prevent portability.

The native communication functions are different and the amount of support the systems provide for communication varies. The Intel supports non-blocking point-to-point communication, some global communication operations, a synchronization function, and IO from any node. The Transtech supports communication between neighboring processors, and only one processor has IO capabilities. The Alex offers blocking point-to-point communication, some global communication operations and IO from any node. All of the machines support C but two different compilers are used for the T805s in the Transtech and Alex and two different versions of the same compiler are used for the i860s in the Intel and the Alex. Despite supporting C, parallel programs would not be portable because of differences in the native communication systems.

Tables 4.1, 4.2 and 4.3 list the BSP parameters for the Intel, Transtech and Alex machines, respectively. The Intel and Transtech systems have similar g and L values. The computation and communication speeds are relatively well matched, the difference being the two systems is the absolute speeds of computation and communication which would be reflected in runtimes. The Alex system has very large g and L values. Computation and communication are not well balanced. The Transtech communication system out performs the Alex communication system, even though it is using the same T805 processors and is configured as a pipeline.

The similarity of high level descriptions offers the possibility of portability but

usually there is something lurking in the assumptions or details which prevents it. The differences at the lower levels increase the complexity of achieving portability. The high level approach taken here, of a common communication library, doesn't require any specific support from the native communication system. The implementor can use whatever support is offered by the native system to produce functions that make use of machine dependencies to give maximum performance.

Summarizing the three machines. The Intel provides lots of communication support and is well balanced. The communication library was relatively easy to implement. The Transtech provide very little communication support but after a careful implementation of the communication library produced a system that shows good balance. The Alex communication system seems to offer reasonable communication support but communication turns out to be very slow and prone to failure. The implementation of the communication library had to try to deal with inadequacies of the native system.

Chapter 5

The Results

The results of using the BSP model to implement and predict the performance of a portable program are presented in this chapter. The first section discusses the effort required to produce a portable implementation, focusing on the communication support required. All of the code line counts given include blank lines and comments. The second section compares the predictions based on the BSP analysis of the parallel sorting algorithm to the actual runtimes of BSP sort. Some measure of the accuracy of the predictions and the sources of error are given, along with suggestions on how the predictions can be improved. The third section compares the runtimes of BSP sort to the runtimes of other sorting programs on similar machines. Overall, the results show that BSP can be used to create a portable implementation without any significant loss of efficiency.

5.1 Portability

There are basically two tasks that need to be considered and are required for portability, the porting of the sequential (non-communication) functions and the implementation of the communication library functions. The sequential code is easily portable, requiring only a compiler for the appropriate programming language and support for system dependent functions, such as, timing functions, input/output and error handling. The functions common to all of the machines required about

400 lines of code and about one month to implement and test.

Portability essentially reduces to the existence of a common communication library on each particular machine, in this case, for the Intel, Transtech and Alex. Eventually it is assumed the manufacturer would supply this library, tuned to give maximum performance for their particular machine. Currently it should be designed and implemented by people that are most familiar with a particular machine. It requires a large initial investment, in time and effort, but will be used by all of the users of the machine.

The high level approach taken here, requiring three global communication functions and synchronization, makes implementation easier. The demands the library makes on the native communication system are less specific. The programmers are free to implement the library in any way that makes the best use of the particular machine and increases performance. Building the global communication libraries required the most work for the portable implementation.

The Intel system provides a lot of communication support and is robust, system failures being rare. The implementation of the communication library on the Intel was relatively simple, since the Intel NX/2 communication library directly supported many of the necessary functions. The implementation of the communication library required about 250 lines of code and about one month to code and test.

The Transtech system provides only communication to adjacent processors. The first task was to implement a point-to-point communication system. This was simplified by the use of a pipeline topology and assumptions, considering the BSP model, about how the system would be used. The communication library was then implemented in terms of the point-to-point functions. Having designed and implemented the point-to-point system with the communication library in mind, it allowed a far simpler implementation that is closer to the hardware than the other systems. This is reflected in the good performance that the library gives. The implementation of the point-to-point communication system and the communication library required about 750 lines of code and about two months to code and test.

The Alex-Trollius system provides a lot of communication support but is slow

and prone to failures. The implementation of communication library required more than expected work as it had to handle exceptions and avoid problems related to the native communication system. Some of the problems encountered included, missing of packets from different sources by a receiver, message truncation without warning by the receiver if the incoming message was larger than the buffer provided, a bug in the multireel function, buffering problems and deadlock due to routing conflicts. These problems required knowledge of the low level operation of Trollius to discover the reasons for the systems failures and find ways of avoiding them. It also made it difficult to track down problems with user programs, which is already a difficult task without the added possibility that system failures might be the cause. The implementation of the communication library required about 450 lines of code and 4 months to code with ongoing testing.

Initially it was difficult to assess which course of action would achieve the best results, to try to work with Trollius or to implement a new point-to-point communication system based on the transputer system. The Alex system is very new and there was very little programming experience using the system to draw on. Afterwards, the exceptionally poor performance of the Trollius system indicated that a lower level implementation would have been better.

The Trollius system has since been superseded by the LAM system [12]. LAM was developed to run on systems that offer more operating system support, like networks of workstations, and currently doesn't run on the Alex. Ongoing work is focusing on tuning Trollius to the Alex system and investigating lower level implementations of the communication library to try to improve performance.

What do the poor BSP numbers for the Alex suggest? The Alex is an unbalanced system and algorithms that have special communication characteristics (very little communication or communication only to adjacent processors) or algorithms that have a much higher computation cost than communication cost should be considered. The parallel sorting algorithm implemented here does not have these properties.

The main advantage offered by the large configurations that are possible using the Alex is the increased amount of system resources, e.g., memory, that can be used

to solve a problem. If a smaller configuration had to solve a problem that required more memory than was available, it would have to start storing partial results to disk. Any sort of file input or output greatly increases runtimes, especially on parallel machines where the processors usually do not have local hard disks and it involves communication with the external system.

Overall, if a fast, robust communication system is available then it is best to use it. This was the approach taken for the Intel. If very little communication support is available then write a fast, robust communication system. This was the approach taken for the Transtech. If a slow, faulty communication system is available, it is probably best to ignore it and write a fast, robust communication system. This was the lesson of the work on Alex.

5.2 Predications

The same parallel sorting program was run on the various configurations of each of the three machines. On each, the value of n , the local number of data items, ranged from 128 up to the maximum that could be accommodated by local memory. All of the graphs of the actual sort runtimes versus the predicted runtimes for each machine configuration can be found in Appendix A.

The sequential, 1 processor, sorting results are shown in Figures A.1, A.5 and A.10 for the Intel, Transtech and Alex machines, respectively. The predictions are based on the computation speed, the problem size and the cost for the sequential sort, $TotalCost = g_P * n \log n$. All of the predictions are close to, or below, the actual sort times. The simplicity of the computation tests used to calculate g_P is somewhat unrealistic and runs at a faster rate than the rate at which the actual sorting program runs.

The predictions for the Transtech Figure A.5 have the largest error, remaining about a constant 24% below the actual runtimes for large values of n . The compiler for the Transtech offers no optimization. It is probably able to handle the small,

simple code of the the test program much better than the large, complex code of the sorting algorithm.

The larger errors for small n on the Alex Figure A.10 reflect the accuracy of the timing available to the i860 nodes. The i860 must poll the T805 to determine clock values which results in a resolution of about 5 milliseconds.

The parallel sorting predictions were made using the BSP parameters for each machine configuration and the problem size substituted into Equation 3.2. The predictions for the Intel for values of $n > 2^{20}$, for 2, 4 and 8 processors were approximately 34% above the actual times with 72% of the error coming from computation predictions and the remaining 28% from the communication predictions. The predictions for the Transtech for values of $n > 2^{19}$, for 4, 8, 12 and 16 processors were approximately 11% above the actual times with 36% due to computation predictions and 64% from communication. Communication times are much less than computation times on the Transtech so errors predicting communication only slightly effect the prediction of the total runtime. The predictions for the Alex for values of $n > 2^{21}$, for 4, 9, 16, 25, 36 and 49 processors were approximately 48% above the actual times with 11% of the error from computation and 89% from communication. Communication times are much larger than computation times on the Alex, so errors predicting communication have a larger effect on the prediction of the total runtime.

The predictions of the individual supersteps match the relative sizes of the actual times for the supersteps on each of the machines. Supersteps 1, 2, 3, 4 are quite small, superstep 5 is a little larger and supersteps 6 and 7 dominate.

In all of the results, the predictions for small values of n , are consistently lower than the actual results. This reflects the choice of g value to predict performance for large n . Recall the condition on the value of g from Chapter 2: the time to route an h -relation is less than or equal to the time to perform gh operations, for all h greater than some h_0 . The predictions start to fail when n drops below the range it was calculated for. The performance of the h -relation, shown in Figures 4.1, 4.2 and 4.3, in the range of interest, is near flat for the Intel, has a slight curve for the

Transtech and has a large curve for the Alex. The value for g_C , used to calculate g , is the average of the h -relation results for h large enough that the expected message size is greater than some constant. This means g_C will be too high for smaller n and too low for larger n . Correspondingly, the value of g will be too low for small n and too high for large n , that is to say, communication predictions based on this value of g will be too low for small n and too high for large n . Allowing g to be a function that depends on the n is one possible solution.

For all of the predictions a value of $\epsilon = 0.5$ was used and the all-to-all, superstep 6, and the final sort, superstep 7, were charged accordingly. This is a worst case analysis since the maximum number of items that a processor ends up with is always less than the number represented by this value, $1.5n$. For large values of n empirical results show a value of $\epsilon \leq 0.1$ is more realistic. Considering the empirical results the use of the larger ϵ value inflates the predictions by about 36% for large n . Providing a more complex analysis to lower the value of ϵ required, would allow for more accurate predictions.

Also, almost all communication functions are overcharged. The analysis considers the maximum amount sent or received by any processor and charges the function as if it were an h -relation of that size. Using different g values for each communication function would avoid this problem.

Tuning the analysis and the BSP model would produce more accurate predictions but would add complexity and reduce portability. The BSP model seeks to predict upper bounds within small constant values of the actual runtimes. The results presented here, using the existing model, satisfy these requirements.

The speedups for BSP sort running on the various configurations of each machine can be found in Table 5.1. The Intel and Transtech show good speedup values. The poor performance of the Alex communication system is apparent in the nearly constant speedup factor independent of the number of processors used. It should be noted that the problem sizes for the large Alex configurations are larger than what the Intel or Transtech systems could handle.

Intel

p	n	Time	Speedup
1	1048576	9.80	-
2	1048576	5.95	1.65
4	1048576	3.51	3.51
8	1048576	1.92	5.10

Transtech

1	1048576	96.79*	-
4	1048576	26.67	3.63
8	1048576	14.68	6.59
12	1572864	16.55	9.03*
16	1048576	8.91	10.87

Alex

1	1048576	11.05	-
4	1048576	7.12	1.55
9	1179648	5.94	2.11*
16	1048576	3.83	2.88
25	819200	3.47	2.44*
36	1179648	4.48	2.80*
49	802816	4.21	1.97*

* indicates estimations

Table 5.1: Speedup Shown by BSP Sort on All Machines

Processors	PSRS	BSP sort
1	8.95	9.80
4	3.02	3.51
8	-	1.92
16	0.90	-

Sorting approximately 2^{20} items.

Table 5.2: Intel Sort Comparisons

Processors	Odd Even merge	Bitonic merge	BSP sort
1	12.07	12.07	21.79
4	4.70	6.78	6.22
8	3.13	2.69	3.11
16	2.37	1.73	2.35

Sorting approximately 2^{18} items.

Table 5.3: Transtech Sort Comparisons

5.3 Efficiency

Comparing BSP sort runtimes to other sorting programs running on similar machines shows that no significant loss of efficiency occurred because of the use of BSF or the portable implementation. To demonstrate that the results of BSP sort are reasonable, the following section lists published sorting results for similar kinds of parallel computers as used here. Comparisons between machines are difficult. Often different numbers of processors are used, different algorithms and the datatype and size handled by the programs vary. All times are given in seconds.

On the Intel iPSC/860 we cite the work of Li *et al.* [37]. They call their approach Parallel Sorting by Regular Sampling (PSRS). The Table 5.2 compares the runtimes of PSRS on 10^6 four byte integers to BSP sort on 2^{20} four byte floats.

On transputer based systems we cite the work of Umland [51] using a network

Processors	Shear sort	Bitonic merge	BSP sort
1	15.00	-	11.07
9	57.00	-	47.31
16	-	167.00	54.32
25	45.00	-	68.86
49	32.00	-	190.41

Sorting approximately 2^{30} items per processor.

Table 5.4: Alex Sort Comparisons

of T800s. The Table 5.3 compares the runtimes of two different sorting algorithms on 250,000 four byte integers to the Transtech version of BSP sort on 2^{18} four byte floats. The Odd Even merge and BSP sort ran on a pipeline topology. The Bitonic merge ran on a hypercube topology.

For the Alex we cite the work of Bauernöppel [7]. The Table 5.4 compares the runtimes of two different sorting algorithms on 10^6 eight byte reals per node to BSP sort on 2^{20} four byte floats per node. Shear sort and BSP sort ran on mesh topologies, Bitonic merge ran on a torus.

The performance of BSP sort is quite reasonable when compared to other sorting results for similar machines, sorting similar amounts of data, with the exception of large Alex configurations. It should also be pointed out that BSP sort was not tuned for any of the particular machines and it offers portability and predictable performance, as well. The use of the BSP model has not caused any significant loss of efficiency in nearly all of the cases.

For large meshes on the Alex, Shear sort outperforms BSP sort by a factor of almost 6, this shows the benefits of considering locality and engaging in only communication with neighboring processors on the Alex machine. This suggests that algorithms that have a very high computation to communication cost ratio should be considered for a BSP implementation. Sorting with a ratio of $n \log n / gn$ is not complex enough. Algorithms that have n^2 or n^3 or higher computation costs

with the same communication cost should be considered, e.g., matrix multiplication.

The problems of the Alex system are beyond the scope of BSP. A basic requirement of the BSP model, the point-to-point communication system, experiences failures. The Alex results highlight the need for portability. The current configuration of the Alex will be changing to improve performance, possibly effecting both the hardware and communication system. For current work to be applicable to the next version of the Alex or to other machines, it must be done in a machine independent, portable fashion.

Chapter 6

Conclusions

The research presented in this thesis shows the utility of the Bulk Synchronous Parallel (BSP) model for parallel computing. BSP offers a simple, practical method for designing, analyzing and implementing portable algorithms that have predictable performance on any machine supporting the BSP model, without any significant loss in efficiency. Valiant's work has shown that BSP is theoretically sound and that algorithms can be designed and analyzed using it. We have given evidence that these algorithms can be implemented in a portable fashion, that the actual runtimes match the prediction runtimes and that the implementation is reasonably efficient.

Portability is one of the main requirements if parallel computing is to enjoy any measure of success. It allows a program to run on different machines and allows a machine which currently runs a program to be upgraded, without requiring the program to be rewritten. The minimal demands BSP makes on a parallel system, mean it can be applied to most systems. The BSP requirement for point-to-point communication completely abstracts the network. Algorithms based on the BSP model are therefore portable and do not depend on any particular network topology. BSP characterizes a parallel computer using a few measured parameters without requiring any detailed analysis or knowledge of the particular system. We opted for a higher level of machine dependency than point-to-point communication. By requiring a communication library of global communication functions it allowed local implementations that were machine dependent, which increased communication

speed. The library could be defined in a BSP manner using point-to-point communication but the complexity of communication suggests using a model that more accurately describes communication, e.g., C^3 , and exploiting machine dependencies to achieve better performance. It requires a large initial investment in time and effort to implement this library but it only has to be done once.

Portability without predictability would not be useful. A BSP analysis of an algorithm's performance is based on the sum of the costs of the supersteps it performs. Computation costs are calculated as in sequential computing, as the number of basic operations performed. Communication costs are scaled to computation using the BSP factor g , the ratio of computation speed to communication speed. Supersteps make a BSP analysis relatively easy to perform. Predictions are accurate within a specific range, depending on how the BSP parameters were measured and chosen, satisfying the model's goal of upper bound predictions within small constant factors of the actual runtimes. If required, there are many ways to improve predictions, for example, allowing g to be a function dependent on the amount of communication or using different g values for different communication functions. BSP stresses simplicity and ease of use over accuracy and strives to avoid complexity. Very accurate predictions require the consideration of too many machine dependent variables. BSP is a compromise which yields good predictions with little complexity.

Portability and predictability without efficiency is a waste of time. Snir had the following to say about portability in [48].

After all, one can always port sequential code on a highly parallel machine just by using one processor, but this is not a very interesting port! (As a matter of fact, even this trivial port is not easy on many highly parallel machines, today.)

Consider such an implementation. It is easily portable, the sequential code can run. It is definitely predictable, the sequential analysis still applies. Is it very efficient? Not in the least.

The simplicity of the BSP model did not greatly effect the performance of the algorithm implemented. The sort times of other sorting programs were offered as

evidence. The BSP model does not hinder the development of efficient algorithms, in fact it can help, by focusing the design on algorithmic bottlenecks, rather than problems that are machine specific. One could start with a BSP implementation, then as simulation and actual execution show machine specific bottlenecks tune the program in portable BSP ways or in machine dependent ways to increase performance. It is reasonable to expect the BSP model to be used in this way.

More research into the use of BSP is suggested by these initial results. The overlapping of communication and computation should be investigated. Only three supersteps are required by the BSP sorting algorithm, one after each of the global communication functions. We used 7 supersteps to simplify analysis, by making each superstep consist entirely of computation or communication. Also, other BSP algorithms should be implemented and tested.

Appendix A

Graphs of Results

A.1 Intel Results

Figures A.1, A.2, A.3 and A.4 show the graphs of the actual runtimes of BSP sort on the Intel iPSC/860 compared to the predictions that were made for 1, 2, 4 and 8 processors, respectively.

Figure A.1: Intel Sort Times, 1 processor

Figure A.2: Intel Sort Times, 2 processors

Figure A.3: Intel Sort Times, 4 processors

Figure A.4: Intel Sort Times, 8 processors

A.2 Transtech Results

Figures A.5, A.6, A.7, A.8 and A.9 show the graphs of the actual runtimes of BSP sort on the Transtech Parastation compared to the predictions that were made for 1, 4, 8, 12 and 16 processors, respectively.

Figure A.5: Transtech Sort Times, 1 processor

Figure A.6: Transtech Sort Times, 4 processors

Figure A.7: Transtech Sort Times, 8 processors

Figure A.8: Transtech Sort Times, 12 processors

Figure A.9: Transtech Sort Times, 16 processors

A.3 Alex Results

Figures A.10, A.11, A.12, A.13, A.14, A.15 and A.16 show the graphs of the actual runtimes of BSP sort on the Alex AVX Series 2 compared to the predictions that were made for 1, 4, 9, 16, 25, 36 and 49 processors, respectively.

Figure A.10: Alex Sort Times, 1 processor

Figure A.11: Alex Sort Times, 4 processors

Figure A.12: Alex Sort Times, 9 processors

Figure A.13: Alex Sort Times, 16 processors

Figure A.14: Alex Sort Times, 25 processors

Figure A.15: Alex Sort Times, 36 processors

Figure A.16: Alex Sort Times, 49 processors

Appendix B

Common Source Code

B.1 BSP Sort Functions

```
/**
psort.h
#define's and globals for parallel sort programs
parallel sort functions
***/

#define E_stack_overflow 2000

#define MIN_SORT 30
#define MAX_STACK 4096

int si, *stack;

#define INT sizeof(int)
#define FLOAT sizeof(float)
#define DOUBLE sizeof(double)

/**
psort.c
02/12/94
Anton
functions needed for parallel sorting
```

```
void local_sort_few(float*, int, int)
    insertion sort

int partition_low(float*, int, int, float)
    partition an array based on a data value [<= | >]

int partition_high(float*, int, int, float)
    partition an array based on a data value [< | >=]

void local_sort_many(float*, int, int)
    local quicksort

void multiple_partition(float*, int , float*, int*)
    multiple partitioning of an array based on an array of values

int parallel_sort(float*, int, int, float*, int, int)
    parallel sort data located on each of the processors

***/

/****
insertion sort
****/
void local_sort_few(float *data, int low, int high) {

    float temp;
    int i, j;

    for (j=low+1; j<=high; j++) {
        temp = *(data+j);
        for (i=j-1; (temp<*(data+i)) && (i>=low); i--)
            *(data+i+1) = *(data+i);
        *(data+i+1) = temp;
    }
} /* end of local_sort_few */

/****
divide an array into two pieces based on pivot value
```

```
[<= | >]
****/
int partition_low(float *data, int low, int high, float pivot) {

    float temp;

    while (low < high) {
        while (*(data+low) <= pivot) {
            if (++low > high) break;
        }
        while (*(data+high) > pivot) {
            if (--high < low) break;
        }
        if (low < high) {
            temp          = *(data+low);
            *(data+low)   = *(data+high);
            *(data+high)  = temp;
        }
    }
    return(high);
} /* end of partition_low */
```

```
/**
divide an array into two pieces based on pivot value
[< | >=]
***/
int partition_high(float *data, int low, int high, float pivot) {

    float temp;

    while (low < high) {
        while (*(data+low) < pivot) {
            if (++low > high) break;
        }
        while (*(data+high) >= pivot) {
            if (--high < low) break;
        }
        if (low < high) {
            temp          = *(data+low);
            *(data+low)   = *(data+high);
            *(data+high)  = temp;
        }
    }
}
```

```
        *(data+high) = temp;
    }
}
return(low);
} /* end of partition_high */
```

/*

local quick sort, partitions data, insertion sorts small pieces
*/

```
void local_sort_many(float *data, int low, int high) {
```

```
    int mid;
    double pivot;
```

```
    si = -2;
    for(;;) {
        if (high-low < MIN_SORT) {
            local_sort_few(data, low, high);
            if (si < 0) break; /* all done want to exit loop */
            low = stack[si];
            high = stack[si+1];
            si -= 2;
        } else {
```

/*

WARNING pivot selection causes a lot of grief
improving this helps the sort a great deal
*/

```
    pivot = (double)data[low];
    pivot += (double)data[((int)low+high)/2];
    pivot += (double)data[high];
    pivot /= 3.0;
```

```
    mid = partition(data, low, high, (float) pivot);
```

```
    if (mid == low) {
        if (data[low] <= (float) pivot) {
            /* unlucky only one data item <= pivot */
            low++;
        } else {
```

```

        /***
        really unlucky math precision problems
        pivot < all data
        partition on known value
        at least one item will be removed
        ***/
        high = p2(data, low, high, data[low]) - 1;
    }
} else if (mid == high) {
    /***
    unlucky pivot == max item in section
    pivot cannot be > all items in the section?
    partition to remove any more items with the same value
    ***/
    high = p2(data, low, high, (float) pivot) - 1;
} else {
    /***
    a nice pivot, two pieces
    push the larger, continue with the smaller
    ***/
    si += 2;
    if (si >= MAX_STACK) error(E_stack_overflow);
    if (mid < (high+low)/2) {
        stack[si] = mid + 1;
        stack[si+1] = high;
        high = mid;
    } else {
        stack[si] = low;
        stack[si+1] = mid;
        low = mid + 1;
    }
}
}
}
} /* end of local_sort_many */

/****
multiple partitioning of the array based on an array of values
modified to deal with number of processors non power of two
****/

```

```

void multiple_partition(data, n, pivots, sizes)
    float *data;
    int n;
    float *pivots;
    int * sizes;
{
    int p1, p2, pi, np;
    int low, high;
    int location[MAX_P];
    int i;

    p1  = -1;
    p2  = p - 1;
    low = 0;
    high = n - 1;
    si  = -4;

    /*****
    pivots | | | | | | | |
           p1  pi  p2
    *****/

    for(;;) {
        pi = (p1 + p2) / 2;
        np = p2 - p1;
        location[pi] = partition(data, low, high, pivots[pi]);
        if (np > 3) { /* both halves need partitioning */
            si += 4; /* push stack */
            if (si >= MAX_STACK) error(E_stack_overflow);
            stack[si+3] = high;
            stack[si+2] = location[pi] + 1;
            stack[si+1] = p2;
            stack[si] = pi;
            high = location[pi];
            p2 = pi;
        } else if (np == 3) { /* only one side needs partitioning */
            if (p1+1 == pi) {
                low = location[pi];
                p1++;
            } else {
                high = location[pi];
                p2--;
            }
        }
    }
}

```



```

    }
    } else { /* np == 2, done this section, check stack for more */
        if (si < 0) break; /* all done, exit loop */
        p1 = stack[si]; /* pop stack */
        p2 = stack[si+1];
        low = stack[si+2];
        high = stack[si+3];
        si -= 4;
    }
}

/**
WARNING the following does not work if
the partitions are singletons or empty
***/
    sizes[0] = location[0] + 1;
    for (i=1; i<p-1; i++)
        sizes[i] = location[i] - location[i-1];
    sizes[p-1] = n - location[p-2] - 1;

} /* end of multpile partition */

/**
parallel sort
sort data spread across many processors
****/
int parallel_sort(data, n, max_n, sample, m, max_m)
    float *data;
    int n;
    int max_n;
    float *sample;
    int m;
    int max_m;
{

    int bytes, step;
    float dividers[MAX_P];
    int sizes[MAX_P];
    int i;

```

```

if (p > 1) {
    for (i=0; i<n; i++) sample[i] = data[my_lrand()%n];
    barrier(the_one);

    bytes = all_to_one(the_one, (char *)sample, m*FLOAT, max_m*FLOAT);
    barrier(the_one);

    if (me == the_one) {
        m = bytes / FLOAT;
        local_sort_many(sample, 0, m-1);
        step = m / p;
        for (i=0; i<p-1; i++) dividers[i] = sample[(i+1)*step];
    }
    barrier(the_one);

    bytes = broadcast(the_one, (char *)dividers, (p-1)*FLOAT);
    barrier(the_one);

    multiple_partition(data, n, dividers, sizes);
    barrier(the_one);

    for (i=0; i<p; i++) sizes[i] *= FLOAT;
    n = all_to_all((char *)data, n*FLOAT, max_n*FLOAT, sizes);
    n /= FLOAT;
    barrier(the_one);
}

local_sort_many(data, 0, n-1);
barrier(the_one);

return(n);
} /* end parallel_sort */

```

B.2 Other Functions

```

/****
copy memory where areas overlap
uses memcpy much faster than for loops
****/

```

2 of/de 2

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

```
void mem_shift(char *to, char *from, int size) {
    int move;

    if (to == from || size == 0) return;
    if (to < from) { /* shifting left */
        move = from - to;
        while (size > 0) {
            if (size < move) move = size;
            memcpy(to, from, move);
            size -= move;
            to += move;
            from += move;
        }
    } else { /* shifting right */
        move = to - from;
        to += size;
        from += size;
        while (size > 0) {
            if (size < move) move = size;
            to -= move;
            from -= move;
            memcpy(to, from, move);
            size -= move;
        }
    }
} /* end mem_shift */

/* @(#)drand48.c 2.2 */
/*LINTLIBRARY*/
/*
 * drand48, etc. pseudo-random number generator
 * This implementation assumes unsigned short integers
 * of at least 16 bits, long integers of at least 32 bits,
 * and ignores overflows on adding or multiplying two
 * unsigned integers. Two's-complement representation is
 * assumed in a few places. Some extra masking is done if
 * unsigneds are exactly 16 bits or longs are exactly 32
 * bits, but so what? An assembly-language implementation
 * would run significantly faster.
 */
```

```

#define N 16
#define MASK ((unsigned)(1 << (N - 1)) + (1 << (N - 1)) - 1)
#define LOW(x) ((unsigned)(x) & MASK)
#define HIGH(x) LOW((x) >> N)
#define MUL(x, y, z) { long l = (long)(x) * (long)(y); \
    (z)[0] = LOW(l); (z)[1] = HIGH(l); }
#define CARRY(x, y) ((long)(x) + (long)(y) > MASK)
#define ADDEQU(x, y, z) (z = CARRY(x, (y)), x = LOW(x + (y)))
#define X0 0x330E
#define X1 0xABCD
#define X2 0x1234
#define A0 0xE66D
#define A1 0xDEEC
#define A2 0x5
#define C 0xB
#define SET3(x, x0, x1, x2)
    ((x)[0] = (x0), (x)[1] = (x1), (x)[2] = (x2))
#define SETLOW(x, y, n)
    SET3(x, LOW((y)[n]), LOW((y)[(n)+1]), LOW((y)[(n)+2]))
#define SEED(x0, x1, x2)
    (SET3(x, x0, x1, x2), SET3(a, A0, A1, A2), c = C)
#define HI_BIT (1L << (2 * N - 1))

static unsigned x[3] = { X0, X1, X2 }, a[3] = { A0, A1, A2 }, c = C;

void my_srand48(long seedval) {
    SEED(X0, LOW(seedval), HIGH(seedval));
}

static void next() {
    unsigned p[2], q[2], r[2], carry0, carry1;

    MUL(a[0], x[0], p);
    ADDEQU(p[0], c, carry0);
    ADDEQU(p[1], carry0, carry1);
    MUL(a[0], x[1], q);
    ADDEQU(p[1], q[0], carry0);
    MUL(a[1], x[0], r);
    x[2] = LOW(carry0 + carry1 + CARRY(p[1], r[0]) + q[1] + r[1] +
        a[0] * x[2] + a[1] * x[1] + a[2] * x[0]);
    x[1] = LOW(p[1] + r[0]);
    x[0] = LOW(p[0]);
}

```

```
}
```

```
long my_lrand48() {  
    next();  
    return (((long)x[2] << (N - 1)) + ((long)x[1] >> 1));  
}
```

```
double my_drnd48() {  
    static double two16m = 1.0 / (1L << N);  
  
    next();  
    return (two16m * (two16m * (two16m * x[0] + x[1]) + x[2]));  
}
```

Appendix C

Intel Communication Library Source Code

```
/*  
comm.h  
29/11/93  
Anton  
  
void error(int errnum)  
    report an error, cleanup and exit  
  
double my_clock()  
    report processor clock value in seconds, machine dependent  
  
void barrier(int root)  
    synchronize all of the processors  
  
int broadcast(int source, char *msg, int bsize)  
    route a message from one processor to all of the others  
  
int all_to_one(int dest, char *msg, int msg_size, int buffer_size)  
    route messages from each processor to one processor  
  
int all_to_all(char *data, int n, int maxn, int *sizes)  
    route different messages from each processor to  
    each of the other processors  
  
void fix(char *data, int n, int maxn, int *msgsize)
```

```
****/

#include <cube.h> /* dclock */

#define E_user 1000
#define E_malloc 1001

#define BROADCAST 10000
#define ALL_TO_ONE 20000
#define ALL_TO_ALL 30000

#define MAX_MEM 16777216
#define MAX_P 8
#define THE_HOST 0

int me, p, the_one;

double tc, tp; /* for collecting execution times */

/**
generic error handling
***/
void error(int err) {
    printf("me %d Error %d\n", me, err);
    exit();
}

/**
generic timing function
***/
double my_clock() {
    double dclock();
    return(dclock());
}
```



```
/**
synchronization function supported by Intel NX system
ignores root
***/
void barrier(int root) {
    gsync();
}

/**
Broadcast
send one message from one processor to all of the others
uses the Intel NX broadcasting facilities
***/
int broadcast(int source, char *msg, int msg_size) {

    if (me == source) {
        csend(BROADCAST, msg, msg_size, -1, 0);
    } else {
        crecv(BROADCAST, msg, msg_size);
        msg_size = infocount();
    }
    return(msg_size);
} /* end of broadcast */

/**
All-to-One
route messages from all the processors to one processor
returns the total size of the data in the buffer
the dest must have enough space to accommodate the incoming data
the dest starts with the message for itself
at the beginning of the message buffer buffer

uses blocking point to point functions
***/
int all_to_one(int dest, char *msg, int msg_size, int buffer_size) {
    int i;

    if (me == dest) {
```

```

    for (i=1; i<p; i++) {
        crecv(ALL_TO_ONE, &msg[msg_size], buffer_size - msg_size);
        msg_size += infocount();
    }
} else {
    csend(ALL_TO_ONE, msg, msg_size, dest, 0);
}
return(msg_size);
} /* end of all_to_one */

```

```

/**
rearrange the data and msgsize arrays for the all_to_all function
data array initially partitioned as: [b1,my,b2,free]
altered to [my,free,b2,b1]
msgsize initially [0, ... ,me, ... ,p-1]
altered to [me,me+1, ... ,p-1,0, ... ,me-1]
reflects order of messages in shifted data array
***/
void fix(char *data, int n, int maxn, int *msgsize) {

```

```

    int endb1;      /* index of end of first block of msgs */
    int startme;    /* index of start of own msg */
    int startb2;    /* index of start of second block */
    int endb2;      /* index of end of second block of msgs */
    int endf1;      /* index of end of free space */
    int startdata;  /* index of start of msgs after shift */
    int sizef1;     /* size of free space */
    int move;       /* size of a shift and temp var */
    int i, j;       /* loop indices */
    char *d1, *d2; /* temp pointers into data array */

```

```

    startme = 0;
    for (i=0; i<me; i++) {
        startme += *(msgsize+i);
    }
    startb2 = startme + *(msgsize+me);
    endb1 = startme - 1;
    endb2 = n - 1;
    endf1 = maxn - 1;
    sizef1 = maxn - n;

```

```
startdata = *(msgsize+me) + sizeof1;

/* data array partitioned as: [b1,my,b2,free] */

while (sizeof1 < endb1+1) {

    /* all of b1 doesn't fit into free space */
    /* must do it in parts */
    /* copy sizeof1 piece of b1 into free space */
    d1 = data + endb2 + 1;
    d2 = data + endb1 + 1 - sizeof1;
    for (i=0; i<sizeof1; i++) {
        *(d1+i) = *(d2+i);
    }
    endf1 -= sizeof1;
    endb1 -= sizeof1;

    /* shift me and b2 to left */
    if (startb2 - sizeof1 < startdata) {
        move = startb2 - startdata;
    } else {
        move = sizeof1;
    }
    d2 = data + startme;
    d1 = d2 - move;
    for (i=0; i<endb2-startme+1; i++) {
        *(d1+i) = *(d2+i);
    }
    startme -= move;
    startb2 -= move;
    endb2 -= move;

} /* end while */

/* all of b1 now fits in free space */
/* copy b1 into end of free space */
d1 = data + endf1 - endb1;
for (i=0; i<endb1+1; i++) {
    *(d1+i) = *(data+i);
}
endf1 -= endb1+1;
```

```

/* need to shift b2 to right */
if (startme < startdata) {
    d1 = data + endf1 - endb2;
    for (i=endb2; i>=startb2; i--) {
        *(d1+i) = *(data+i);
    }
    endf1 = startdata - 1;
}

/* need to shift my block left */
if (startme > 0) {
    d2 = data + startme;
    for (i=0; i<*(msgsize+me); i++) {
        *(data+i) = *(d2+i);
    }
}

/* now data array partitioned as: [my,free,b2,b1] */

/* shift msgsize array to reflect altered data array */
for (i=0; i<me; i++) {
    move = *msgsize;
    for (j=0; j<p-1; j++) {
        *(msgsize+j) = *(msgsize+j+1);
    }
    *(msgsize+p-1) = move;
}

} /* end of fix */

```

/**

All-to-All

route different messages from each processor to each of the others

data is sent and received using the same buffer

uses non-blocking point to point communication functions

allows two sends and a receive to be started before

waiting for the receive

*/

```

int all_to_all(char *data, int n, int maxn, int *sizes) {

```

```
char *start_free, *start_data;
int sid1, sid2, rid, i;

fix(data, n, maxn, sizes);
start_free = data + sizes[0];
start_data = start_free + maxn - n;

sid1 = isend(ALL_TO_ALL, start_data, sizes[1], (me+1)%p, 0);
for (i=2; i<p; i++) {
    rid = irecv(ALL_TO_ALL, start_free, start_data-start_free);
    start_data += sizes[i-1];
    sid2 = isend(ALL_TO_ALL, start_data, sizes[i], (me+i)%p, 0);
    msgwait(sid1);
    sid1 = sid2;
    msgwait(rid);
    start_free += infocount();
}
rid = irecv(ALL_TO_ALL, start_free, start_data-start_free);
msgwait(sid1);
msgwait(rid);
start_free += infocount();

return(start_free - data);
} /* end of all_to_all */
```

Appendix D

Transtech Communication Library Source Code

```
/**
comm.h
#include for transputer libraries
#define global constants
global variables used for communication
***/

#include <channel.h> /* ChanOut,ChanIn */
#include <process.h> /* ProcAlloc,ProcTime */
#include <stdlib.h> /* abort,EXIT_SUCCESS */
#include <semaphor.h> /* SemAlloc,SemWait */
#include <string.h> /* memcpy,memmove */

#define THE_HOST 0

#define RIGHT 0
#define LEFT 1
#define NLINK 2
#define HEADER_SIZE 8
#define PACKET_SIZE 8192 /* size in bytes */
#define N_COMM_BUFF 3 /* # of buffers per cRecv process */
#define MAXQ 4 /* max msg queue size */

/* error messages */
```

```
#define E_SemAlloc 1
#define E_malloc 2
#define E_ProcAlloc 3
#define E_msg_too_big 4
#define E_recv_pending 5
#define E_send_to_self 6
#define E_failure 7
#define E_pivots 8
#define E_stack_overflow 9

#define TO_MASK 15
#define LAST_PACKET 32
#define BARRIER 64

struct msg_queue {
    int dest;
    char *start;
    int size;
    Semaphore *done;
    int route;
};

struct send_queue {
    Semaphore *access;
    Semaphore *sn; /* number of msgs in queue */
    Semaphore *snf; /* number of free spaces in queue */
    int head;
    int tail;
    struct msg_queue info[MAXQ];
};

struct data_queue {
    Semaphore *access;
    char *start;
    int size;
    int recvd;
    int nmsg;
    Semaphore *done;
};

struct send_queue qSend[NLINK]; /* message queues */
Semaphore *sdata[NLINK]; /* signal changes in data space size */
```

```
struct data_queue qData; /* recv data queue */

Semaphore *BarrierSem; /* used for barrier function */

int me;      /* local processor id */
int p;       /* number of processors in pipe */
int the_one; /* special processor in pipe */

/**
system dependent functions
***/

/**
return clock value in seconds
***/
double my_clock() {
    return((double)ProcTime()/1000000.0);
}

/**
wait a specified length of time, seconds
***/
void my_wait(int how_long) {
    /**
    how_long in tenths of seconds
    depending on priority clock tick is
        1 micro (high) or
        64 micro (low)
    convert accordingly
    ***/
    if (ProcGetPriority() == 1)
        how_long /= 64; /* low priority */

    ProcWait((int)(how_long * 1000000.0));
} /* end my_wait */

#include "bcomm.c"
#include "acomm.c"
```



```
/**
bcomm.c
10/12/94
Anton
Packetizing version
basic low level communication operations
***/

void setup_comm(Channel*, Channel*);
    /* setup communication */

void Send(int, char*, int, Semaphore*, int);
    /* nonblocking send */

Semaphore *Recv(char*, int, int);
    /* nonblocking receive */

int Wait_Recv(Semaphore*);
    /* wait for receive to complete */

void cSend(Process*, struct send_queue*, Channel*, int);
    /* process that manages sending messages */

void cRecv(Process*, Channel*, Semaphore*, char*, int);
    /* process that manages receiving and routing messages */

void error(int);
    /* error handler */

****
allocate semaphores and communication processes
start communication processes
****/
void setup_comm(Channel *In, Channel *Out) {
    int i;
    Process *pOut[NLINK], *pIn[NLINK];
    char *buff;
```

```
/* allocate semaphores */
for (i=0; i<NLINK; i++) {
    if ((qSend[i].access = SemAlloc(1)) == NULL)
        error(E_SemAlloc);
    if ((qSend[i].sn = SemAlloc(0)) == NULL)
        error(E_SemAlloc);
    if ((qSend[i].snf = SemAlloc(MAXQ)) == NULL)
        error(E_SemAlloc);
    qSend[i].head = 0;
    qSend[i].tail = -1;
}

if ((qData.access = SemAlloc(1)) == NULL)
    error(E_SemAlloc);
qData.start = NULL;
qData.size = 0;
qData.recvd = 0;
qData.nmsg = 0;
qData.done = NULL;

for (i=0; i<NLINK; i++)
    if ((sdata[i] = SemAlloc(0)) == NULL)
        error(E_SemAlloc);

if ((BarrierSem = SemAlloc(0)) == NULL)
    error(E_SemAlloc);

/* allocate communication processes */
i = 0;
if (me < p-1) {
    pOut[RIGHT] = ProcAlloc(cSend, 512, 3,
        &qSend[RIGHT], Out[i], RIGHT);

    if (pOut[RIGHT] == NULL) error(E_ProcAlloc);

    buff = (char *) malloc(N_COMM_BUFF*PACKET_SIZE);
    if (buff == NULL) error(E_malloc);

    pIn[RIGHT] = ProcAlloc(cRecv, 512, 4,
        In[i], sdata[RIGHT], buff, RIGHT);
```

```
    if (pIn[RIGHT] == NULL) error(E_ProcAlloc);
    i++;
}
if (me > 0) {
    pOut[LEFT] = ProcAlloc(cSend, 512, 3,
        &qSend[LEFT], Out[i], LEFT);

    if (pOut[LEFT] == NULL) error(E_ProcAlloc);

    buff = (char *) malloc(W_COMM_BUFF*PACKET_SIZE);
    if (buff == NULL) error(E_malloc);

    pIn[LEFT] = ProcAlloc(cRecv, 512, 4,
        In[i], sdata[LEFT], buff, LEFT);

    if (pIn[LEFT] == NULL) error(E_ProcAlloc);
}

/* start communication processes */
if (me < p-1) {
    ProcRunHigh(pOut[RIGHT]);
    ProcRunHigh(pIn[RIGHT]);
}
if (me > 0) {
    ProcRunHigh(pOut[LEFT]);
    ProcRunHigh(pIn[LEFT]);
}

} /* end of setup */

/****
user interface for sending messages
****/
void Send(dest, start, bsize, done, route)
    int dest;        /* process id of destination */
    char *start;     /* pointer to message */
    int bsize;       /* size of message, bytes */
    Semaphore *done; /* semaphore to signal when done sending */
    int route;       /* originating or routing message */
}
```

```
{
    struct send_queue *qS;
    struct msg_queue *qM;
    int to;

    to = dest & TO_MASK;

    if (to == me) error(E_send_to_self);
    qS = &qSend[(int)(to < me)]; /* left or right queue */
    /* add msg to end of queue */
    SemWait(qS->snf); /* wait for space in the queue */
    SemWait(qS->access);
    qS->tail = (qS->tail + 1) % MAXQ;
    qM = &qS->info[qS->tail];
    qM->dest = dest;
    qM->start = start;
    qM->size = bsize;
    qM->done = done;
    qM->route = route;
    SemSignal(qS->sn); /* signal addition to queue */
    SemSignal(qS->access);
} /* end of Send */

/****
user interface for receiving messages
****/
Semaphore *Recv(start, bsize, nmsg)
    char *start; /* pointer to buffer */
    int bsize; /* size of buffer, bytes */
    int nmsg; /* number of message to receive */
{
    Semaphore *recv_done;

    if ((recv_done = SemAlloc(0)) == NULL) error(E_SemAlloc);
    SemWait(qData.access);
    if (qData.start != NULL) error(E_recv_pending);
    qData.start = start;
    qData.size = bsize;
    qData.recvd = 0;
    qData.nmsg = nmsg;
```

```
    qData.done = recv_done;
    SemSignal(qData.access);
    SemSignal(sdata[RIGHT]); /* signal change to size of data space */
    SemSignal(sdata[LEFT]);
    return(recv_done);
} /* end of Recv */
```

```
/*
```

```
user interface for waiting for messages to arrive
```

```
*/
```

```
int Wait_Recv(Semaphore *done) {
    int size;

    SemWait(done);
    free(done);
    SemWait(qData.access);
    size = qData.recvd;
    qData.start = NULL;
    qData.size = 0;
    qData.recvd = 0;
    qData.nmsg = 0;
    qData.done = NULL;
    SemSignal(qData.access);
    return(size);
} /* end of Wait_Recv */
```

```
/*
```

```
process to manage output of messages over one of the links
```

```
*/
```

```
void cSend(P, qS, link, dir)
    Process *P;          /* pointer to this process */
    struct send_queue *qS; /* pointer to send request queue */
    Channel *link;        /* pointer to link sending on */
    int dir;              /* direction sending in */
{
    struct msg_queue *qM;
    int header[2];
    int size;
```

```
char *start;
Semaphore *done;
int route;

/* AWS avoid warnings */
P = P; dir = dir;

for (;;) {
    /* get msg info from queue */
    SemWait(qS->sn); /* waits if no msg pending */
    SemWait(qS->access);
    qM = &qS->info[qS->head];
    qS->head = (qS->head+1) % MAXQ;
    header[0] = qM->dest;
    size = qM->size;
    start = qM->start;
    done = qM->done;
    route = qM->route;
    SemSignal(qS->snf);
    SemSignal(qS->access);

    if (route == 1) { /* message being routed */
        header[1] = size;
        ChanOut(link, header, HEADER_SIZE);
        if (size > 0) ChanOut(link, start, size);
    } else { /* message originating here */
        if (size <= PACKET_SIZE) { /* only one packet or none */
            header[0] |= LAST_PACKET;
            header[1] = size;
            ChanOut(link, header, HEADER_SIZE);
            if (size > 0) ChanOut(link, start, size);
        } else { /* packetize */
            header[1] = PACKET_SIZE;
            while (size > PACKET_SIZE) {
                ChanOut(link, header, HEADER_SIZE);
                ChanOut(link, start, PACKET_SIZE);
                start += PACKET_SIZE;
                size -= PACKET_SIZE;
            }
            header[0] |= LAST_PACKET;
            header[1] = size;
            ChanOut(link, header, HEADER_SIZE);
        }
    }
}
```

```
        ChanOut(link, start, size);
    }
}
    if (done != NULL) SemSignal(done);
}
} /* end of cSend */
```

/*****

process to manage input and routing of messages
arriving on a particular link

****/

```
void cRecv(P, link, sdata, buff, dir)
```

```
    Process *P;          /* pointer to this process */
    Channel *link;        /* pointer to link receiving on */
    Semaphore *sdata;     /* pointer to receive message queue */
    char *buff;           /* pointer to private buffer */
    int dir;              /* direction receiving from */
```

```
{
```

```
    int header[2];
    char *start;
    int signal;
    Semaphore *buff_free;
    int ibuff;
```

```
    /* AWS avoid warnings */
```

```
    P = P; dir = dir;
```

```
    if ((buff_free = SemAlloc(N_COMM_BUFF)) == NULL)
        error(E_SemAlloc);
```

```
    ibuff = 0;
```

```
    signal = 0;
```

```
    for (;;) {
```

```
        ChanIn(link, header, HEADER_SIZE);
```

```
        /* received an error msg */
```

```
        if (header[1] < 0) error(header[1]);
```

```
        if ((header[0] & TO_MASK) == me) {
```

```
            /* message for local processor */
```

```
            /* use space indicated by qData */
```

```
if (header[1] > 0) {
    SemWait(qData.access);
    while (qData.size < header[1]) {
        /* not enough space OR not expecting a msg */
        SemSignal(qData.access);

        SemWait(sdata); /* wait until qData changes */

        SemWait(qData.access);
    }
    start = qData.start;
    qData.start += header[1];
    qData.size -= header[1];
    /* AWS dec count if end of msg */
    if (header[0] & LAST_PACKET) {
        if (--qData.nmsg == 0) signal = 1;
        /* SemSignal(LoadSem[dir]); */
    }
    SemSignal(qData.access);

    ChanIn(link, start, header[1]);

    SemWait(qData.access);
    qData.rcvd += header[1];
    if (signal > 0) {
        SemSignal(qData.done);
        signal = 0;
    }
    SemSignal(qData.access);
} else {
    if (header[0] & BARRIER) { /* barrier message */
        SemSignal(BarrierSem);
    } else {
        SemWait(qData.access);
        if (--qData.nmsg == 0) SemSignal(qData.done);
        SemSignal(qData.access);
        /* SemSignal(LoadSem[dir]); */
    }
}
} else {
    /* forward message */
```



```
/* use communication buffers */
if (header[1] > 0) {
    SemWait(buff_free);
    ibuff = (ibuff+1) % N_COMM_BUFF;
    ChanIn(link, &buff[ibuff*PACKET_SIZE], header[1]);
    Send(header[0], &buff[ibuff*PACKET_SIZE], header[1],
        buff_free, 1);
} else {
    Send(header[0], NULL, 0, NULL, 1);
}
}
}
} /* end of cRecv */
```

```
/*****
function to deal with errors
uses the basic communication function
sends error code msg to THE_HOST
THE_HOST then prints it
then exits this function never returns
****/
void error(int error_num) {
    Semaphore *done;

#ifdef IO
    int pid;
#endif

    done = SemAlloc(0);
#ifdef IO
    if (error_num < 0) {
        pid = (error_num*-1) / 1024;
        error_num = (error_num*-1)%1024;
    } else {
        pid = me;
    }
    printf("Error %d %d\n", pid, error_num);
#else
    if (error_num > 0)
```

```
        error_num = (me*1024 + error_num) * -1;
        Send(TH_HOST, NULL, error_num, done, 0);
    #endif
    if (done != NULL) /* SemAlloc succeeded */
        SemWait(done);

    exit_terminate(error_num);
} /* end of function error */
```

/**

acom.c

19/10/93

Anton

Collective communication functions

assuming the network is a pipeline

*/

```
int broadcast(int, char*, int);
int all_to_one(int, char*, int, int);
int all_to_all(char*, int, int, int*);
void barrier(int);
```

/**

Broadcast

one processor, source, sends the same msg to all other processors

the others receive the message and forward it as necessary

*/

```
int broadcast(int source, char *msg, int bsize) {
    Semaphore *recv_done, *send_done;

    if ((send_done = SemAlloc(0)) == NULL) error(E_SemAlloc);

    if (me == source) {
        /* send msg in all possible directions, right and left */
        if (me < p-1) Send(me+1, msg, bsize, send_done, 0);
        if (me > 0) Send(me-1, msg, bsize, send_done, 0);
        SemWait(send_done);
```

```
    if (me > 0 && me < p-1) SemWait(send_done);
} else {
    /* receive one msg */
    recv_done = Recv(msg, bsize, 1);
    bsize = Wait_Recv(recv_done);
    /* send msg on as necessary, right or left */
    if (me > source) {
        if (me < p-1) {
            Send(me+1, msg, bsize, send_done, 0);
            SemWait(send_done);
        }
    } else {
        if (me > 0) {
            Send(me-1, msg, bsize, send_done, 0);
            SemWait(send_done);
        }
    }
}
free(send_done);
return(bsize);
} /* end of broadcast */
```

/*

All-to-One

all processors send a msg to one processor, dest

*/

```
int all_to_one(int dest, char *msg, int msg_size, int buffer_size) {
    Semaphore *recv_done, *send_done;

    if ((send_done = SemAlloc(0)) == NULL) error(E_SemAlloc);

    if (me == dest) {
        recv_done = Recv(msg+msg_size, buffer_size-msg_size, p-1);
        msg_size += Wait_Recv(recv_done);
    } else {
        Send(dest, msg, msg_size, send_done, 0);
        SemWait(send_done);
    }
    free(send_done);
    return(msg_size);
}
```

```
} /* end of all_to_one */
```

```
/**
```

```
All-to-All
```

```
each processor sends a different message to  
each of the other processors
```

```
sends occur in a modulus offset order
```

```
input array stores messages in node order [0, ..., p-1]
```

```
*/
```

```
int all_to_all(char *data, int n, int maxn, int *size) {
```

```
    char *data_start, *free_start;
```

```
    int msgsize, to;
```

```
    int i;
```

```
    Semaphore *recv_done, *send_done;
```

```
    int startme, startb2, startdata;
```

```
    int endb1, endb2, endf1, sizef1, move;
```

```
    int j;
```

```
    int slow;
```

```
    if ((send_done = SemAlloc(0)) == NULL) error(E_SemAlloc);
```

```
/**
```

```
rearrange data and msgsize arrays
```

```
data array initially partitioned as: [b1,my,b2,free]
```

```
altered to [my,free,b2,b1]
```

```
msgsize initially [0, ... ,me, ... ,p-1]
```

```
altered to [me,me+1, ... ,p-1,0, ... ,me-1]
```

```
reflects order of messages in shifted data array
```

```
*/
```

```
    startme = 0;
```

```
    for (i=0; i<me; i++) startme += size[i];
```

```
    startb2    = startme + size[me];
```

```
    endb1      = startme - 1;
```

```
    endb2      = n - 1;
```

```
    endf1      = maxn - 1;
```

```
    sizef1     = maxn - n;
```

```
    startdata = size[me] + sizef1;

/* data array partitioned as: [b1,my,b2,free] */
while (sizef1 < endb1+1) {
    /* all of b1 doesn't fit into free space */
    /* must do it in parts */
    /* copy sizef1 piece of b1 into free space */
    memcpy(data+endb2+1, data+endb1+1-sizef1, sizef1);
    endf1 -= sizef1;
    endb1 -= sizef1;

    /* shift me and b2 to left */
    if (startb2 - sizef1 < startdata) {
        move = startb2 - startdata;
    } else {
        move = sizef1;
    }
    memcpy(data+startme-move, data+startme, endb2-startme+1);
    startme -= move;
    startb2 -= move;
    endb2 -= move;
} /* end while */

/* all of b1 fits now fits in free space */
/* copy b1 into end of free space */
memcpy(data+endf1-endb1, data, endb1+1);
endf1 -= endb1+1;

/* need to shift b2 to right */
if (startme < startdata) {
    mem_shift(data+startdata, data+startb2, endb2-startb2+1);
    endf1 = startdata - 1;
}

/* need to shift my block left */
if (startme > 0) memcpy(data, data+startme, size[me]);

/* now data array partitioned as: [my,free,b2,b1] */

/* shift size array to reflect altered data array */
for (i=0; i<me; i++) {
    move = size[0];
```

```
    for (j=0; j<p-1; j++) size[j] = size[j+1];
    size[p-1] = move;
}

slow = (int) (n > p*PACKET_SIZE); /* simple load control */

free_start = data + size[0];
data_start = free_start + maxn - n;

if (slow == 0)
    recv_done = Recv(free_start, data_start - free_start, p-1);

for(i=1; i<p; i++) {

    if (slow == 1)
        recv_done = Recv(free_start, data_start - free_start, 1);

    msgsize = size[i];
    to = (me+i) % p;
    if (msgsize <= PACKET_SIZE) {
        Send(to|LAST_PACKET, data_start, msgsize, send_done, 1);
        data_start += msgsize;
        SemWait(send_done);
        SemWait(qData.access);
        qData.size += msgsize;
        SemSignal(qData.access);
        SemSignal(sdata[RIGHT]);
        SemSignal(sdata[LEFT]);
    } else {
        while (msgsize > PACKET_SIZE) {
            Send(to, data_start, PACKET_SIZE, send_done, 1);
            msgsize -= PACKET_SIZE;
            data_start += PACKET_SIZE;
            SemWait(send_done);
            SemWait(qData.access);
            qData.size += PACKET_SIZE;
            SemSignal(qData.access);
            SemSignal(sdata[RIGHT]);
            SemSignal(sdata[LEFT]);
        }
        Send(to|LAST_PACKET, data_start, msgsize, send_done, 1);
        data_start += msgsize;
    }
}
```

```
    SemWait(send_done);
    SemWait(qData.access);
    qData.size += msgsize;
    SemSignal(qData.access);
    SemSignal(sdata[RIGHT]);
    SemSignal(sdata[LEFT]);
}
if (slow == 1) {
    free_start += Wait_Recv(recv_done);
    barrier(the_one);
}
}
if (slow == 0) {
    free_start += Wait_Recv(recv_done);
    barrier(the_one);
}

free(send_done);
return(free_start - data);
} /* end of all_to_all */
```

/**

Barrier

works in a special way

special zero length messages sent to the source.

low level recv recognizes a barrier message

and signals a global semaphore

when everyone has logged in

the source broadcasts a signal to start everyone back up

*/

```
void barrier(int source) {
```

```
    int i;
```

```
    if (p > 1) {
```

```
        if (me == source) {
```

```
            /* wait for all processors to check in */
```

```
            for(i=1; i<p; i++) SemWait(BarrierSem);
```

```
        } else {
```

```
            /* report in to the source processor */
```

```
            Send(source|BARRIER, NULL, 0, NULL, 0);
```

```
    }  
    /* send out signal to continue */  
    broadcast(source, NULL, 0);  
  }  
} /* end of barrier */
```


Appendix E

Alex Communication Library Source Code

```
/*  
Alex comm.h  
machine dependent stuff  
#includes  
#defines  
global variables  
  
my_clock  
my_wait  
exit_clean  
error  
read_route  
  
communication library  
****/  
  
#include <castreq.h>  
#include <net.h>  
  
#define ALL 111  
#define BARRIER 10000  
#define BROADCAST 20000  
#define ALL_TO_ONE 30000  
#define ALL_TO_ALL 10000000
```

```
#define E_user 1000
#define E_malloc 1001
#define E_buffer 1002

#define MAX_COUNT 1000
#define PACKET_SIZE 4096 /* 16384 also available */

int maxx, maxy, nturns;
char route[MAX_P];

int me, p, the_one;
int si, *stack;
extern int errno; /* standard error code */

/* counters for events in global communication functions */
int barrier_count = 1;
int broadcast_count = 1;
int all_to_one_count = 1;
int all_to_all_count = 1;

int my_flag; /* used in all_to_all to help control network load */

double my_clock(void);
void my_wait(int);
void exit_clean(int);
void error(int);
void read_route(void);
void barrier(int);
int broadcast(int, char*, int);
int all_to_one(int, char*, int, int);
int atoa_recv(int, char*, char*);
int atoa_send(int, int, char*, int*, int*);
int all_to_all(char*, int, int, int*, int*);

/**
my_clock
returns clock time in seconds
the i860 has no clock, it polls the T805
***/
#ifdef i860
```

```
#define WAIT_FIX 104000
double rtttime(int);
double my_clock() {
    /* get time from T805 on the current node */
    return(rtttime(getnodeid()));
}
#else
double ttime();
double my_clock() {
    return(ttime());
}
#endif

/**
my_wait
wait a specified number of seconds
***/
#ifdef i860
/**
i860 doesn't have a sleep function
processor busy waits for the specified length of time
with WAIT_FIX how_long specifies seconds
***/
void my_wait(int how_long) {
    how_long *= WAIT_FIX;
    while (how_long > 0) {
        my_wait(-1);
        how_long--;
    }
} /* end my_wait */
#else
my_wait(int how_long) {
    sleep(how_long)
}
#endif

/**
cleanup and exit
```

must remove rcast definition

***/
void exit_clean(int errno) {

/* remove rcast definition */

/* the_one sends rmelt message to other processors */

if (me == the_one) rmelt(LOCAL, ALL);

/* Detach from Trollius kernel */

kexit(errno);

} /* end of exit_clean */

/**

print error message and exit

***/
void error(int err) {

printf("%d Error %d\n", me, err);

exit_clean(err);

} /* end of error */

/**

read all_to_all routing information from input file

store in a char array

< 64 send

> 64 recv

***/
void read_route() {

int i, j;

struct nmsg md;

int tmp;

md.nh_flags = 0;

md.nh_type = 0;

md.nh_length = p * sizeof(char);

md.nh_msg = (char *) malloc(md.nh_length);

/* read and distribute routing table for all_to_all */

if (me == the_one) {

```

scanf("%d", &nturns);
*md.nh_msg = (char)nturns;
broadcast(the_one, md.nh_msg, sizeof(char));

for (i=0; i<p; i++) {
    if (i == the_one) {
        for (j=0; j<nturns; j++) {
            scanf("%d", &tmp);
            if (tmp >= 100) tmp -= 36;
            *(route+j) = (char)tmp;
        }
    } else {
        md.nh_node = i;
        md.nh_event = i;
        for (j=0; j<nturns; j++) {
            scanf("%d", &tmp);
            if (tmp >= 100) tmp -= 36;
            *(md.nh_msg+j) = (char)tmp;
        }
        if (nsend(&md) < 0) error(errno);
    }
}
} else {
    broadcast(the_one, md.nh_msg, sizeof(char));
    nturns = (int) *md.nh_msg;

    md.nh_event = me;
    md.nh_msg = route;
    if (nrecv(&md) < 0) error(errno);
}
free(md.nh_msg);
} /* end read_route */

/**
Barrier
synchronize the processors
uses Trollius multicast/reel facilities
rcast definition in main
***/
void barrier(int root) {

```

```

struct nmsg md;

if (p > 1) {
    /* multicast tree defined for the_one only */
    if (root != the_one) error(E_failure);

    /* recv ready signal from all the nodes */
    md.nh_event = BARRIER + barrier_count;
    md.nh_type = 0;
    md.nh_length = 0;
    md.nh_node = ALL;
    md.nh_flags = WREEL;

    if (me == the_one) {
        if (nrecv(&md) < 0) error(errno);
    } else {
        if (nsend(&md) < 0) error(errno);
    }

    /* broadcast a start signal to all the nodes */
    md.nh_event += 1000;
    md.nh_flags = 0;
    if (me == the_one) {
        if (nsend(&md) < 0) error(errno);
    } else {
        if (nrecv(&md) < 0) error(errno);
    }
}

if (++barrier_count > MAX_COUNT) barrier_count = 1;
} /* end of barrier */

```

/**

Broadcast

route a message from one processor to all of the others
uses the Trollius multicast facilities
based on rcast definition in main
limited only one processor, the_one, can act as source
allowing any processor to act as source would require
dynamic use of rcast which is a very slow operation
***/

```
int broadcast(int source, char *msg, int msg_size) {
    struct nmsg md;

    if (p == 1) return(msg_size);

    /* multicast tree defined for the_one only */
    if (source != the_one) error(E_failure);

    md.nh_node    = ALL;
    md.nh_event    = BROADCAST + broadcast_count;
    md.nh_flags    = 0;
    md.nh_type     = 0;
    md.nh_length   = msg_size;
    md.nh_data[0] = msg_size;
    md.nh_msg      = msg;
    if (me == source) {
        if (nsend(&md) < 0) error(errno);
    } else {
        if (nrecv(&md) < 0) error(errno);
        /* check if receiver truncated message */
        if (md.nh_length != md.nh_data[0]) error(E_failure);
    }

    if (++broadcast_count > MAX_COUNT) broadcast_count = 1;

    return(md.nh_length);
} /* end of broadcast */
```

/**

All-to-One

route messages from all the processors to one

uses trollius network layer functions

but manages its own packetization

*/

```
int all_to_one(int dest, char *msg, int msg_size, int buffer_size) {
    struct nmsg md;
    int i, k;
    char *msg_p[MAX_P];
    int size;
    char tmp[PACKET_SIZE];
```

```
if (p == 1) return(msg_size);
```

```
/**
```

```
do our own packetization
```

```
assume everyone sending same sized messages
```

```
if variable sized messages would need to first send sizes
```

```
store messages in node order in msg array
```

```
*/
```

```
md.nh_node = dest;
```

```
md.nh_event = ALL_TO_ONE + all_to_one_count;
```

```
md.nh_flags = 0;
```

```
if (me == dest) {
```

```
    size = p * msg_size;
```

```
    if (buffer_size < size) error(E_buffer);
```

```
    md.nh_msg = tmp;
```

```
    msg_p[0] = msg;
```

```
    for(i=1; i<p; i++) msg_p[i] = msg_p[i-1] + msg_size;
```

```
    if (me != 0) memcpy(msg_p[me], msg, msg_size);
```

```
    while (msg_size < size) {
```

```
        md.nh_length = PACKET_SIZE;
```

```
        md.nh_type = 0;
```

```
        if (nrecv(&md) < 0) error(errno);
```

```
        memcpy(msg_p[md.nh_type], tmp, md.nh_length);
```

```
        msg_p[md.nh_type] += md.nh_length;
```

```
        msg_size += md.nh_length;
```

```
    }
```

```
    } else {
```

```
        md.nh_type = me;
```

```
        md.nh_length = PACKET_SIZE;
```

```
        md.nh_msg = msg;
```

```
        size = msg_size;
```

```
        while (size > PACKET_SIZE) {
```

```
            if (nsend(&md) < 0) error(errno);
```

```
            md.nh_msg += PACKET_SIZE;
```

```
            size -= PACKET_SIZE;
```

```
        }
```

```
        if (size > 0) {
```

```
            md.nh_length = size;
```

```
            if (nsend(&md) < 0) error(errno);
```



```
    }
}

    if (++all_to_one_count > MAX_COUNT) all_to_one_count = 1;
    return(msg_size);
} /* end of all_to_one */

/**
send one message during all-to-all
uses tranport layer Trollius functions
manages data arrays
***/
int atoa_send(int to, int i, char *data, int *msg_size, int *dest) {
    int j, k;
    struct nmsg m;

    if (to < 0 || to > p) error(E_failure);
    m.nh_flags    = my_flag;
    m.nh_type     = 0;
    m.nh_node     = to;
    m.nh_event    = ALL_TO_ALL + to*100000 + i*1000 + all_to_all_count;
    m.nh_length   = msg_size[to];
    m.nh_data[0] = m.nh_length;

    /* find start of the msg */
    m.nh_msg = data;
    for (j=0; dest[j] != m.nh_node; j++)
        m.nh_msg += msg_size[dest[j]];

    if (tsend(&m) < 0) error(errno);

    /* fix dest array */
    for (k=j; k<p-1; k++) dest[k] = dest[k+1];
    dest[p-1] = to;

    /* shift data to close gap */
    mem_shift(data+m.nh_length, data, m.nh_msg-data);

    return(m.nh_length);
} /* end of atoa_send */
```

```

/**
recv one message during all-to-all
uses transport layer Trollius function
***/
int atoa_recv(int i, char *data, char *buff) {
    struct msg m;

    m.nh_flags = my_flag;
    m.nh_type = 0;
    m.nh_event = ALL_TO_ALL + me*100000 + i*1000 + all_to_all_count;
    m.nh_length = data - buff;
    m.nh_msg = buff;

    if (trecv(&m) < 0) error(errno);

    /* check trecv did not truncate message */
    if (m.nh_length != m.nh_data[0]) error(E_buffer);

    return(m.nh_length);
} /* end of atoa_recv

/**
data [0,1,...,p-1,free]
n number of data bytes
maxn total number of bytes in data array
msg_size[p] size in bytes of message to processor i
dest[p] order of messages in data array

all_to_all events
|-|--|--|---|
| | | | |
| | | |count (0 < MAX_COUNT < 1000)
| | | |
| | |turn (0 to MAXP-1)
| | |
| |id (0 to MAXP-1)
| |
|type (ALL_TO_ALL)

```

```
***/  
int all_to_all(char *data, int n, int maxn, int *msg_size) {  
    char *init, *buff;  
    int i, j, k, to;  
    int dest[MAX_P];  
  
    if (p == 1) return(n);  
  
    init = data;  
  
    for (i=0; i<p; i++) dest[i] = i;  
  
    /* shift data all the way to the right */  
    mem_shift(data+maxn-n, data, n);  
    data += maxn - n;  
  
    /* copy my message to front */  
    i = 0;  
    for (j=0; dest[j] != me; j++) i += msg_size[dest[j]];  
  
    memcpy(init, data+i, msg_size[me]);  
    buff = init + msg_size[me];  
  
    /* shift data to fill gap */  
    mem_shift(data+msg_size[me], data, i);  
    data += msg_size[me];  
  
    /* fix dest array */  
    for (k=j; k<p-1; k++) dest[k] = dest[k+1];  
    dest[p-1] = me;  
  
    /* allow buffering for large messages on large meshes */  
    if (p>25 && n>p*PACKET_SIZE) {  
        my_flag = 0;  
    } else {  
        my_flag = NOBUF;  
    }  
  
    for (i=0; i<nturns; i++) {  
        /**/  
        route entry < MAX_P send then recv  
        route entry > MAX_P recv then send
```

```
    ***/
    to = (int) route[i];
    if (to < MAX_P) {
        data += atoa_send(to, i, data, msg_size, dest);
        buff += atoa_recv(i, data, buff);

    } else {
        buff += atoa_recv(i, data, buff);
        data += atoa_send(to-64, i, data, msg_size, dest);
    }
}
if (++all_to_all_count > MAX_COUNT) {
    all_to_all_count = 1;
}
return(buff - init);
} /* end of all_to_all */
```

Bibliography

- [1] F. Adolhassan, R. Drefenstedt, J. Keller, W.J. Paul, and D. Scheerer. On the physical design of PRAMs. *Computer Journal*, 36(8):756-762, Dec. 1993.
- [2] R. Alasdair, A. Bruce, J.G. Mills, and A.G. Smith. CHIMP version 2.0 user guide. Technical report, University of Edinburg, Mar. 1994.
- [3] Alex Informatique Inc. *Alex-Trollius Library Reference in C*, Mar. 1992.
- [4] Alex Informatique Inc. *AVX Series 2 Parallel Computer System Technical Description*, 1993.
- [5] D.H. Bailey. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputing Review*, 4(8):54-55, Aug. 1991.
- [6] E. Barszcz. One year with an iPSC/860. In *IEEE COMPCON Spring*, pages 283-288, 1991.
- [7] F. Bauernöppel. Basic algorithms for the AVX parallel computer. Technical report, Carleton University, 1994.
- [8] J.R. Beers, Jr., R. Leibensperger, M. Braner, and D. Fielding. Transputer performance issues using the Trollius operating system. In H. Zedan, editor, *Real-Time Systems with Transputers*, pages 35-50. IOS Press, 1990.
- [9] R. Berendorf and J. Helin. Evaluating the basic performance of the Intel iPSC/860 parallel computer. *Concurrency: Practice and Experience*, 4(3):223-240, May 1992.

- [10] R.H. Bisseling and WF McColl. Scientific computing on bulk synchronous architectures. Technical Report 836, Department of Mathematics, University of Utrecht, Dec. 1993. Short version appears in Proceedings of the 13th IFIP World Computer Conference. Volume I 1994. B Pehrson and I Simon, Eds., Elsevier, 1994, p509-514.
- [11] G.E. Blelloch, C.E. Leiserson and B.M. Maggs, C.G. Plaxton, S.J. Smith, and M. Zagha. A comparison of sorting algorithms for the connection machine CM-2. In *ACM SPAA 91*, pages 3-16, July 1991.
- [12] G. Burns, R. Daoud, and J Vaigl. LAM: An open cluster environment for MPI. Technical report, Ohio Supercomputer Center, May 1994.
- [13] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing - A paradigm for transportable software. Technical Report TR-36-94, Center for Research in Computing Technology, Harvard University, 1994. A version of this report appears in the Proceedings of the 28th Annual Hawaii International Conference on systems Sciences, Jan. 1995.
- [14] L.A. Crowl. How to measure, present, and compare parallel performance. *IEEE Parallel and Distributed Technology*, Spring:9-25, 1994.
- [15] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eiken. LogP: Towards a realistic model of parallel computation. Technical report, Computer Science Division, University of California, Berkeley, 1993. A version of this report appears in the Proceedings of 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming, May 1993, San Diego, CA.
- [16] J. Darlington and H.W. To. Building parallel applications without programming. Technical report, Department of Computing, Imperial College, London, 1993.

- [17] F. Dehne, A. Fabri, and A. Rau-Chaplin. Scalable parallel geometric applications for coarse grained multicomputers. In *ACM Symposium on Computational Geometry*, pages 298–307, 1993.
- [18] T.H. Dunigan. Performance of the Intel iPSC/860 and Ncube 6400 hypercubes. *Parallel Computing*, 17:1285–1302, 1991.
- [19] J. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings 10th Annual ACM Symposium on Theory of Computing*, pages 114–118, 1978.
- [20] MPI Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Apr. 1994. Available from <http://www.mcs.anl.gov/mpi>.
- [21] W.D. Frazer and A.C. McKellar. Samplesort: A sampling approach to minimal storage tree sorting. *JACM*, 17(3):496–507, July 1970.
- [22] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V.S. Sunderam. PVM 3 user's guide and reference manual. Technical Report ORNL/TM-12187, Oak Ridge National Laboratory, Sept. 1994.
- [23] A.V. Gerbessiotis and L.G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, pages 251–267, Aug. 1994.
- [24] H.H. Goldstine. *The Computer from Pascal to von Neumann*. Princeton University Press, Princeton, NJ, 1972.
- [25] S.E. Hambrusch and A.K. Khokhar. C^3 : An architecture-independent model for coarse-grained parallel machines. Technical report, Department of Computer Science, Purdue University, Dec. 1993.
- [26] A. Heddaya and A.F. Fahmy. OS support for portable bulk synchronous parallel programs. Technical Report BU-CS-94-012, Boston University, Computer Science Department, 1994.

- [27] J. Helin and R. Berrendorf. Analyzing the performance of message passing MIMD hypercubes: A study with the Intel iPSC/860. In *ACM International Conference on Supercomputing*, pages 376–385, June 1991.
- [28] W.L. Hightower, J.F. Prins, and J.H. Reif. Implementations of randomized sorting on large parallel machines. In *ACM SPAA 92*, pages 158–167, 1992.
- [29] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1):10–15, 1962.
- [30] G. Hristescu. Parallel triangulation of a set of points for coarse grained multi-computers. Technical report, Rutgers University, Oct. 1994.
- [31] J.S. Huang and Y.C. Chow. Parallel sorting and data partitioning by sampling. In *IEEE COMPSAC 83*, pages 627–631, Nov. 1983.
- [32] INMOS Limited. *The Transputer Databook*, 1992.
- [33] Intel Corporation. *iPSC User's Guide*, 1985.
- [34] Intel Corporation. *i860 Manual Set*, 1991.
- [35] B.H.H. Juurlink and H.A.G. Wijshoff. Experiences with a model for parallel computation. In *12th ACM Symposium on Principles on Distributed Computing*, pages 87–96, 1993.
- [36] H. Li and K.C. Sevcik. Parallel sorting by overpartitioning. In *ACM SPAA 94*, pages 46–56, June 1994.
- [37] 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, 1993.
- [38] J.K. Littlejohn. Dynamic bulk synchronous parallelism. Technical Report EPCC-SS92-16, Edinburgh Parallel Computing Centre, 1992.

- [39] W.F. McColl. BSP programming. In G. Bluelloch, M. Chandy, and S. Jaggannathan, editors, *Proceedings of the DIMACS Workshop on Specification of Parallel Algorithms*. American Mathematical Society, May 1994. To appear.
- [40] R. Miller. A library for bulk synchronous parallel programming. Technical report, Oxford University Computing Laboratory, 1994.
- [41] R. Miller and J. Reed. The Oxford BSP library users' guide version 1.0. Technical report, Programming Research Group, University of Oxford, 1993. This paper plus several others about the Oxford BSP Library and the library itself are available by ftp from: `ftp.comlab.ox.ac.uk` in `/pub/Packages/BSP`.
- [42] R. Ponnusamy, R. Thakur, A Choudhary, K. Velamakanni, Z. Bozkus, and G. Fox. Experimental performance evaluation of the CM-5. *Journal of Parallel and Distributed Computing*, 19:192-202, 1993.
- [43] A. Rau-Chaplin. *On Parallel Data Structures and Parallel Geometric Applications for Multicomputers*. PhD thesis, Carleton University, Ottawa, Canada, Nov. 1992.
- [44] J.H. Reif and L.G. Valiant. A logarithmic time sort for linear size networks. *JACM*, 31(1):60-76, Jan. 1987.
- [45] M. Sawyer. Bulk synchronous parallelism on the Meiko Computing Surface. Master's thesis, University of Edinburgh, Apr. 1992.
- [46] D.B. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, pages 23-50, Dec. 1990.
- [47] D.B. Skillicorn. Models for practical parallel computation. *International Journal of Parallel Programming*, 20(2):133-128, 1991.
- [48] M. Snir. Issues and directions in scalable parallel computing. In *12th ACM Symposium on Principles on Distributed Computing*, pages 21-28, 1993.

- [49] T.M. Stricker. Supporting the hypercube programming model on mesh architectures (A faster sort for iWarp tori). In *ACM SPAA 92*, pages 148–157, 1992.
- [50] Transtech Parallel Systems. *PARAstation User Manual*, 1992.
- [51] T. Umland. Parallel sorting revisited. *Parallel Computing*, 20(1):115–124, 1994.
- [52] L.G. Valiant. A scheme for fast parallel communication. *SIAM Journal of Computing*, 11:350–361, 1982.
- [53] L.G. Valiant. Bulk-synchronous parallel computers. In M. Reeve and S.E. Zenith, editors, *Parallel Processing and Artificial Intelligence*, chapter 2, pages 15–22. Wiley, 1989.
- [54] L.G. Valiant. A bridging model for parallel computation. *CACM*, 33(8):103–111, 1990.
- [55] L.G. Valiant. General purpose parallel architectures. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 18, pages 945–971. North Holland, Amsterdam, 1990.
- [56] J. von Neumann. *Collected Works*, volume V. Pergamon Press, Oxford, England, 1961. A.H. Taub, editor.
- [57] W. Zimmerman and W. Löwe. An approach to machine-independent parallel programming. In B. Buchberger and J. Volkert, editors, *Parallel Processing: CONPAR 95-VAPP VI*, pages 277–288, 1994.

END

2 1-1 1-9 5

FIN