1

PM-1 3 1/2 x 4 PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

PRECISIONSM RESOLUTION TARGETS

Developing Parallel GIS Applications on the Alex AVX II Computer

By
David Roytenberg

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

**Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario**

September 1996

**© Copyright
1996, David Roytenberg**

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

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 accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

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 propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-612-19483-3

Canada

Name David Roytenberg

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

0984
SUBJECT CODE

U-M-I

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
Journicism 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 0488
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
Science 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0298
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

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

PHILOSOPHY, RELIGION AND THEOLOGY

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

SOCIAL SCIENCES

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

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0321
Asia, Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Weed Technology 0746
Biology
General 0306
Anatomy 0287
Botanics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0349
Limnology 0793
Microbiology 0410
Molecular 0307
Neurosciences 0317
Oceanography 0416
Physiology 0423
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0760

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0996

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Palaeobotany 0345
Palaeozoology 0426
Palaeontology 0418
Palaeozoology 0985
Polynology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0547
Education 0350
Hospital Management 0769
Human Development 0738
Immunology 0982
Medicine and Surgery 0544
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

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

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

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0525
Physiological 0989
Psychiatry 0349
Psychometrics 0432
Social 0431

The undersigned hereby recommend to
the Faculty of Graduate Studies and Research
acceptance of the thesis.

Developing Parallel GIS Applications on the Alex AVX II Computer

submitted by

David Roytenberg

Dr. Evangelos Kranakis
(Director, School of Computer Science)

Dr. Jörg-Rüdiger Sack
(Thesis Supervisor)

Carleton University
September 1996

Abstract

Geographical Information Systems (GIS) can place great computational demands on sequential computer systems. However the use of parallel computing for GIS has been slow to develop due to the increased difficulty of parallel programming. We describe the implementation of tools to facilitate the development of parallel GIS applications. These tools include a parallel cellular automata driver and a simulator. The driver allows cellular automata applications to be developed for the parallel platform transparently. That is to say that the developer need not take the parallelism into account in the course of the application development. We implemented a demonstration GIS application for the cell driver and two stand-alone applications on the Alex AVX II, which is a message passing parallel computer with a configurable topology. Substantial improvements in performance are demonstrated as increasing numbers of parallel processors are used.

Acknowledgments

I would like to thank my thesis supervisor Dr. Jörg-Rüdiger Sack whose vision has made this research possible and whose dedication to teaching and research have served as an example to me. His help and comments have greatly improved this thesis.

I would also like to thank the other members of the Carleton GIS group, especially Anil Maheshwari, Doron Nussbaum, David Hutchinson and Mark Lanthier for many ideas and discussions. Thanks to Nashib Qadri for useful discussions on parallel fire modeling and to Zhengping Wang for *test driving* the the Cell Driver.

Thanks to Wolfgang Bitterlich for making the CanBEIS code available and for supplying test data, and to Keith Clarke for the sequential Fire Model code and for answering questions very promptly by e-mail.

Thanks also to John Neilson for much help in the course of development of XAS, to Peter Choynowski for system support and making his "inkmesh" script available for use with XAS. Also much appreciation to Patrick Morin who helped by solving Parasol issues as they arose during the development of XAS.

Anton Saarimaki was generous with his time and experience when I was learning Trollius and during the work on CanBEIS and XAS.

I feel great appreciation to the National Science and Engineering Research Council (NSERC) and Almerco Inc. for their financial support during the last two years. Without their help, I would not have had the opportunity to do this work.

Thanks especially to my wife, Dora Benbaruk for her patience and support.

Contents

Abstract	iii
Acknowledgments	iv
1 Scope of the Research	1
2 Background to the Work	4
2.1 Parallel Computing	4
2.1.1 Parallel Computers	9
2.1.2 Development Platform for the Work	11
2.1.3 Theoretical Models of Parallel Computation	13
2.1.4 Strategies for Parallel Solution of Computational Problems	15
2.2 Geographical Information Systems	17
2.3 Cellular Automata	19
2.4 The Applications	21
2.4.1 CanBEIS	21
2.4.2 Overhead of Parallel Computation in CanBEIS	24
2.4.3 Clarke Fire Model	27
2.4.4 Strategy for Fire Model Parallelization	28
2.5 The Cell Driver	29
2.5.1 Introduction to the Cell Driver	29
2.5.2 Neighbourhood Modeling (NEMO) System	30
2.5.3 Criteria for Beneficial Parallel Implementations of Cellular Automata	32

2.5.4	The Simple Fire Model of Brinch Hansen	34
3	Design Issues	36
3.1	CanBEIS	36
3.2	Fire Model	40
3.3	The Cell Driver	43
3.3.1	Partitioning of Data Among the Processors	43
3.3.2	Neighbour Data Exchange	46
3.3.3	GIS Issues	47
4	Cell Driver: Goals, Design and Development	49
4.1	Management of Generality	49
4.2	Components of the Cell Driver	52
4.2.1	Overview	52
4.2.2	Application and Driver Levels on the Host and Node	53
4.2.3	Compute_Map	55
4.2.4	Initial Setup	56
4.2.5	The transition function	58
4.2.6	Convert to Display Image	59
4.2.7	The setup for nextIteration function	60
4.3	Integration with other NEMO Components	60
4.4	Demonstration Application	61
5	XAS: Goals, Design and Development	65
5.1	Motivation for the Project	65
5.2	Background to the XAS Project	67
5.2.1	Simulation of Parallel Computers	68
5.2.2	The Alex AVX II Parallel Computer	68
5.2.3	The Parasol Modelling Environment	70
5.2.4	SAS Simulator	70
5.3	Overview of the XAS Program	73
5.3.1	XAS Data Structures	73

5.3.2	Basic XAS Functions	73
5.3.3	XAS Tasks	74
5.4	Trollius Operating System Components and their Emulation in XAS	75
5.4.1	Emulation of Trollius Kernel Level Messaging in XAS	75
5.4.2	Emulation of Physical Messaging in XAS	79
5.4.3	The Emulation of Data Link Messaging in XAS	79
5.4.4	Data Link and Buffer Daemon emulation in XAS	82
5.4.5	Emulating the Trollius Network Layer Code in XAS	83
5.4.6	Emulation of Trollius Routing in XAS	84
5.4.7	The Emulation of the Trollius Cast System in XAS	85
5.4.8	Synchronizing Processes on the Simulated i860 and T805 of an XAS Compute Node	87
5.4.9	Summary of Trollius Processes Emulated in XAS	89
5.5	Conclusion and suggestions for further work on XAS	91
6	Some Results of the Work	92
6.1	A Note on Measuring Parallel Performance	92
6.2	CanBEIS	94
6.3	Clarke Fire Model	95
6.4	Brinch Hansen Fire Model and Cell Driver	101
6.5	Conclusions and Further Work	104
	Bibliography	106

List of Figures

2.1	Diagram of Trollius Communication Layers	12
2.2	Data Movement on a 16 Node Square Mesh of Processors	25
3.1	Graphical Display of Total Gas Emission in CanBEIS	37
3.2	Partitioning an Arbitrary Raster Over an Arbitrary Mesh	45
3.3	Examples of Cell Neighbourhoods that Require Data Exchange with 4 and 8 Neighbours	47
4.1	Behaviour of the <i>Default_Compute_Map</i> Function	56
4.2	The Display Output of the Demonstration Application	63
5.1	Kernel Communication in Trollius and XAS	76
5.2	Virtual Circuits in Trollius and XAS	78
5.3	XAS System Tasks	90
6.1	Speedups for Parallel CanBEIS	96
6.2	Performance of the Modified Sequential Version of the Clarke Fire Model	98
6.3	Performance of the Parallel Fire Model Running on one Node	98
6.4	Performance of the Parallel Fire Model Running on 64 Nodes	99
6.5	Graph of Absolute Speedup vs. Number of Iterations on 64 Nodes . . .	100
6.6	Graph of Relative Speedup Versus Number of Iterations on 64 Nodes	101
6.7	The Perimeter Cells to be Exchanged for a 3 by 3 Neighbourhood . . .	103
6.8	Speedups of the Demo Application for Two Different Raster Sizes . . .	104

Chapter 1

Scope of the Research

This thesis describes an investigation of the feasibility of running Geographical Information System (GIS) applications on the Alex AVX II parallel computer. The initial goal of this work was primarily to demonstrate the feasibility of porting existing sequential GIS applications to a parallel platform, and secondly, to show that useful speedups can be achieved while doing so. As this work proceeded other subsidiary goals emerged and were pursued.

One of these goals was to gain a greater understanding of the cost of communication in the Alex AVX II parallel computer. This knowledge provides greater insight into the communication costs introduced by the move to this parallel platform. More generally, the identification of the communication and computation costs of parallel applications provides a guide to the sort of applications that can benefit in terms of performance from the use of multiprocessing capabilities. It also helps us to identify those applications for which the use of parallel techniques is not likely to be worthwhile.

In the course of the work, two stand-alone GIS applications were implemented in order to study the behavior of real applications in the parallel environment and to explore effective strategies for parallelization. In addition, we formulated the following objectives in the course of the work:

- Understand the requirements for general tools

- GIS tools

- Application tools

Application tools that have been implemented include XAS, a simulator for the Alex AVX II. (see Chapter 5) and *vis*, a tool for visualization of parallel program performance. (*Vis* was developed by Craig MacDonald in the course of the work for his Master's thesis [35]). One of the parallel programs developed as part of this thesis (CanBEIS: see Section 2.4.1) was used by Craig in the development of *vis* as a test example.

One of the GIS tools that we set out to implement was a cellular automata driver (cell driver). In the context of the cell driver we implemented a simple fire model based on one described in [25]. This model is introduced in Section 2.5.4. A second cell driver application (an earthquake modeling system) is being developed by Zhengping Wang.

Another line of research that flows from this work is the search for ways to get better performance from the AVX system, by using knowledge of the system's characteristics to improve the design of applications, as well as by seeking ways to improve the system software that comes with the AVX II. A great deal of insight into the system software was gained in the course of the work on XAS described in Chapter 5.

The first application which we implemented (parallel CanBEIS) is an example of data parallelism. The second application, (the Clarke Fire Model) shows that in some cases, functional parallelism is straightforward to implement.

The third application is an idealized fire model implemented in conjunction with the cell driver, as a cellular automaton. It is based on an algorithm described by Per Brinch Hansen in [25]. It was developed primarily as a test application for the cell driver (CD), which represents a considerably more complex and more general approach to applying parallel computation to the field of Geographical Information Systems, and to spatial modeling in general.

The cell driver attempts to abstract out the common elements of cellular automata applications as they must be implemented on a distributed memory parallel

machine. This allows the developer of parallel cellular automata applications to focus more on the algorithm and less on the parallelism. Because there are applications of cellular automata in many domains, it is anticipated that the cell driver will be used to develop a number of cellular automata applications within the Carleton GIS group's Neighbourhood Modeling (NEMO) system (see Section 2.5.2 for an introduction to NEMO). Currently, an earthquake modeling application is being developed by Zhengping Wang, another Master's student in our group, using the cell driver as a platform.

A good part of the effort described in this thesis was devoted to the development of XAS, the detailed simulator for the Alex AVX II. One reason for the development of the simulator was to provide a means of tracing the message passing behaviour of a program that is in the process of development. It also makes it possible to debug programs that are destined for the parallel platform using source code debuggers that are only available on the sequential platform. The design and development of the simulator is described in Chapter 5 of this thesis.

In the course of the work on the simulator, some useful information was obtained concerning the time cost of message passing on the AVX II. This information is in turn useful for assessing the feasibility of parallelizing a given application if the volume of data that is to be exchanged relative to the amount of computation is known.

The work on the time cost of messaging, known as the AVX timing project was undertaken in conjunction with Linda Cummins and Patrick Morin during the summers of 1995 and 1996, respectively. Using the numbers obtained from this work, it is hoped to introduce realistic performance prediction into the simulator mentioned above.

Chapter 2 provides the necessary background to the work discussed in the rest of the thesis. Chapter 3 looks at the interesting design issues that arose in the course of the development of the parallel applications. Chapter 4 provides an overview of the design of the Cell Driver component of the NEMO project. Chapter 5 discusses the design and development of XAS, a simulator for the Alex AVX II parallel computer. Chapter 6 presents some measurements of the performance of the parallel programs developed for this thesis.

Chapter 2

Background to the Work

2.1 Parallel Computing

This research and development is about the application of parallel computing. The development work was done on a parallel computer. In order to discuss this work it is necessary first to discuss the utility of parallel computing. Then we state what we mean by parallel computing and finally we talk about the alternatives available in the way of parallel hardware and the particular parallel machine that we used for this work.

Parallel hardware is discussed in Section 2.1.1. Here we discuss some of the research that is being done in this fast growing field.

Parallel computing has gained a wide following in recent years and is a focus of research at many institutions. This is partly because the falling cost of hardware has brought multiprocessing capabilities within reach of more institutions than was formerly the case. Another reason is the growing list of areas in which parallel processing has begun to produce interesting results.

This variety can be seen by looking at the web site of the National Centre for High Performance Computing (NCHPC) in the United States. The NCHPC groups government agencies and universities engaged in a wide variety of research and development.

The consortium *research thrusts*¹ show some of the areas in which parallel computing work is being pursued today. These general areas encompass some of the current investigation being conducted by the Paradigm group here at Carleton.

- **Geophysics and Astrophysics**

1. Climate, Weather, Ocean Model
2. Astrophysics and Planetary Science
3. Ocean Acoustics
4. Plasma Physics
5. Petroleum Geophysics
6. Ecological and Social/Economic Modeling
7. Environmental Modeling

- **Chemical and Biological Science**

1. Molecular Modeling
2. Neural Sciences
3. Gene Sequencing
4. Biological Databases
5. Biomedical Imaging

- **Quantum Systems**

1. Condensed Matter Physics
2. Particle Physics and Quantum Chromodynamics
3. Quantum Chemistry

- **Computational Fluid Dynamics (CFD)**

¹This list was taken from the *Research Thrusts* web page of the NCHPC at URL <http://www.nchpc.lcs.mit.edu/Info/Overview/thrusts.html>

- 1. CFD and Turbulence**
 - 2. Aerodynamic Design**
 - 3. Structures**
 - 4. Shock Physics**
 - 5. Electromagnetics**
 - 6. Discrete Elements**
 - 7. Chemical Engineering**
 - 8. Combustion**
- **Computational Algorithms**
 - 1. Scalable Algorithms**
 - 2. Computer Vision and Signal Processing**
 - 3. Finite Element and Grid Generation**
 - **Architectures and Software Tools**
 - 1. Parallel Operating Systems and Programming Tools**
 - 2. Distributed Computing**
 - 3. Performance Evaluation**
 - 4. Visualization**
 - 5. Data Management and Networking**
 - 6. Parallel Architectures**

Members of the NCHPC include the National Science Foundation Supercomputer Centers, including

- **Cornell Theory Center (CTC)**
- **National Center for Atmospheric Research (NCAR)**

- National Center for Supercomputing Applications (NCSA)
- Pittsburgh Supercomputing Center (PSC)
- San Diego Supercomputer Center (SDSC)

Universities in the consortium include

- MIT's Laboratory for Computer Science (LCS)

At MIT the Large-scale Parallel Software Group web page describes their goals: *Research in the Large-scale Parallel Software Group is focused on software for parallel and distributed computing systems. Building efficient, portable software for parallel and distributed platforms is notoriously difficult; our research is aimed at making this easier. Our research can be broadly classified into three areas: making parallel and distributed programs easier to write; building efficient runtime systems for parallel and distributed programs; and bridging the gap between parallel and distributed systems to allow a common hardware base to be used for a broad range of applications.* ².

- University of Colorado's Center for Applied Parallel Processing (CAPP)

The Center for Applied Parallel Processing (CAPP) at the University of Colorado at Boulder has as its goal: *...the solution of Grand Challenge research problems through the deployment of scalable computing and the development of scalable algorithms for such systems.* ³

- Boston University
- Harvard University
- University of California System

Other US Government Agencies include the National Oceanic and Atmospheric Administration of the Department of Commerce, where supercomputing is used for

²<http://www.psg.lcs.mit.edu>

³This information is taken from the CAPP web page <http://www.mcb.cs.colorado.edu/home/capp/whatis.htm>

weather forecasting and environmental modeling. Environmental modeling is one of the areas of application of the work being done by the Carleton GIS group.

Other work in parallel computing covers many areas at both the system and application levels.

For example, at Adelaide University in Australia, optimization of parallel computing is being pursued by means of a functional programming language called *Adl* [1]. This group began by doing performance studies of the CM-5, which is their target machine [19].

Other groups are focusing on automatic parallelization of procedural languages [38].

In Amsterdam the Computer Aided Migration of Applications System (CAMAS) project is using machine modeling and performance prediction in an effort to facilitate the automatic parallelization of existing programs.

Our own work focuses on the development of tools for parallel spatial information systems (SIS) and in particular geographical information systems (GIS) with the primary objective of *transparent parallelism*. We are developing *middle-ware* that is higher level than the parallel operating system, but which enables application developers to write spatial applications for the parallel machine in much the same way that they would write a sequential application.

Like the Adelaide group we have devoted some time to measuring the performance characteristics of our parallel platform, the Alex AVX II. The AVX timing project studied the fine grained behaviour of message passing on the AVX II. The performance results for several parallel applications developed on the AVX II are presented in Chapter 6.

Because of the greater complexity and difficulty of parallel development, the effort is pursued only in fields where standard computers are insufficiently powerful. The applications listed above have in common a great appetite for computing resources. Spatial Information Systems are among those applications for which existing computing resources place constraints on the size, accuracy and timeliness of the results obtainable. Spatial Information Systems are introduced in Section 2.2.

2.1.1 Parallel Computers

The increasing interest in parallel computing is due in part to the fact that the cost of hardware has been falling and to the consequent availability of a number of practical parallel machines.

A *parallel computer* is one which can execute more than one instruction at the same time (in parallel). A parallel computer is called *synchronous*, if at each time step, all processors simultaneously execute an instruction. Otherwise it is called *asynchronous*. A parallel computer is of the *SIMD* variety (single instruction multiple data stream) if all processors execute the same instruction at the same time, possibly using different data for the instructions. A computer is of the *MIMD* variety (multiple instruction multiple data stream) if processors may execute different instructions simultaneously. Parallel processors may communicate via a shared memory, or via a connection network.

All of the parallel algorithms developed in this thesis are designed to run on an asynchronous *MIMD* parallel computer in which the processors communicate via a connection network.

Commercial high performance computers encompass a variety of models, of which *MIMD* parallel computers are an important and growing instance.⁴ These models include

- parallel vector processors
- shared memory symmetric multiprocessors
- workstations connected with a powerful switch
- message passing *MIMD*
- SIMD

Vector processors are represented by the Cray T-90, Fujitsu VPP and the NEC SX-4. These are the systems originally referred to as *supercomputer*.

⁴The following taxonomy of parallel computers first appeared in [28]

Shared memory symmetric multiprocessors are typically based on the Unix operating system and are represented by

- the Silicon Graphics Challenge and Power Challenge series
- the DEC 82xx
- the Cray CS6400
- the Sequent Symmetry and NUMA-Q series
- the Convex Exemplar line.

This architecture is becoming very prevalent in the mid-performance range for HPC systems. Tera Computer is a new entrant in this field at the very high-end with their multi-threaded architecture. A new trend is to connect multiple high performance workstations with a fast switch. The IBM SP-2 is an example of a commercial system based on this concept.

Message passing *MIMD* systems tend to be proprietary architectures which can be scaled to very large numbers of processors. Most of these systems are based on commodity microprocessors. Some suppliers have developed custom processors and/or used more exotic processors. Systems of this type include

- the AVX Series II and III from Alex Informatique
- the Cray T3E
- the nCube 2S family
- the Meiko CS-2
- the Parsytec GC/Powerplus family.

Some systems, such as the Mercury Race products, allow for heterogeneous systems to be configured.

SIMD systems have not been much in favor. Nonetheless Cambridge Parallel Processing has been successfully marketing their DAP system for image processing and similar applications for many years.

2.1.2 Development Platform for the Work

In this section we discuss the particular parallel machine that was used for the work described in this thesis. As we have seen there is less standardization among the architectures of parallel machines than there is in the realm of sequential machines. There is greater complexity in the hardware and system software of those machines that are available. Therefore it is necessary to look carefully at the development platform before discussing the work.

The AVX series II parallel computer is manufactured by Alex Informatique based in Montréal, Canada. The largest AVX II has 64 standard nodes and several additional nodes providing external access. A standard node consists of two processors, one Intel *i860* vector processor, used for computations, with between 32 and 64MB of DRAM, and one processor (Inmos *T805* transputer), used primarily for communication, with between 8 and 16MB of DRAM.

The AVX II is a distributed memory *MIMD* parallel machine. It can be reconfigured into a variety of topologies. Such topologies include meshes, hypercubes and trees. It can also be put in nonstandard application specific configurations. The communication among the internal nodes and between the host, a Sun Microsystems SPARC 20, and the internal nodes of the AVX is based on the Trollius operating system designed by Ohio State University. For an overview of Trollius see [11]. The Trollius version for the AVX is introduced in [13] and [12].

A feature of the Trollius system is its multi-layered communication subsystem. Users can modify a number of parameters in the network layer including the route that data packets take while moving between processors and the amount and type of buffering provided by the intermediate processors along a data packet's route. For the performance characteristics of Trollius communication see [5].

Trollius is structured as a kernel process and a number of servers or daemons. The kernel process provides a local message passing interface and the daemons communicate with each other using this interface. The daemons themselves implement the higher level message passing functions of Trollius. They include servers which manage message buffering, routing, and multi-casting as well as data link input and output daemons that channel data going into and out of each node. Process scheduling is

Figure 2.1: Diagram of Trollius Communication Layers

not managed by Trollius, but is assumed to be managed by an underlying system. In the AVX, the unix system provides scheduling on the host, while a simple kernel does scheduling on the Transputer. On the *i860* only one process at a time can run and thus only a simple loader is needed.

Trollius offers a number of different layers of message passing. (See Figure 2.1)

The foundation of this communication hierarchy is the Trollius kernel, which provides a local interprocess messaging capability, and the physical communication layer which implements nearest neighbour messaging between Trollius nodes.

The data link layer provides inter-nodal messaging between nearest neighbours using a simple interface analogous to the local messaging provided by the kernel. The *data link layer* actually allows senders to reach a process anywhere in the AVX, but beyond the nearest neighbour, the network layer is used.

The *network layer* adds automatic routing and large message packetization to the data link layer services. While the data link layer has a fixed maximum packet size, the network layer can send messages of arbitrary length by packetizing them at the sending end and reassembling the packets at the receiving end.

Finally, the *transport layer* adds a connection oriented messaging service built on top of the network layer. When a network message is sent, the packets are sent to

the destination node by the system where they are buffered if necessary and await a network receive request. Because of this there is a risk that the network can become full of packets that are never received. The transport layer eliminates this risk by first exchanging small signals between sender and receiver before the sending of the message itself begins. Both sending and receiving tasks block until the transmission of the message is complete.

Development work for this thesis was done on the five Alex AVX II machines presently installed at the School of Computer Science, at Carleton University; two 16-node, one 32-node, and two 64-node machines, each having up to 80 MB of internal memory per node. The total memory is therefore approximately 12 GB. When handling very large problems, the availability of this much memory means that applications that could not fit their data into memory in a sequential platform can be run entirely in memory on a parallel machine. In addition to the gains from greater numbers of processors, gains due to additional memory are expected to produce a dramatic improvement in the performance of such applications.

Distributed memory parallel computation is a focus of our research and that of many others all over the world. This is partly because commodity microprocessors can now be connected into networked distributed memory multiprocessors at costs much lower than that of the specially engineered uniprocessors that would be required to achieve the same rate of computation.

As the number of processors in a distributed memory multiprocessor grows, rates of computation may be achieved that are beyond the capacity of any conceivable uniprocessor. However the achievement of these rates depends on the appropriate decomposition of problems so as to balance the work among the available processors and so as to minimize the volume of communication among them.

2.1.3 Theoretical Models of Parallel Computation

Well before the advent of real parallel machines, the potential of parallel computation to decrease the lower bounds on the time cost of algorithms was recognized. In the absence of real machines, theoreticians worked with models that expressed the

power of multiple processors, but used a rather idealistic model of communication. For example, a great deal of theoretical work was done using a model in which an arbitrary number of processors share a common memory. This model is known as the Parallel Random Access Machine (PRAM). A good introduction to the PRAM model and many PRAM algorithms is found in [33]. A particularly good example of the great power of PRAM techniques is found in [3].

The PRAM model has the virtue that it resembles the standard Random Access Machine model used for sequential machines. However it was soon recognized that ignoring communication will often produce algorithms that run quickly on a PRAM, but run slowly on a real parallel machine. Other models have attempted to deal with this by modeling the connection network.

These models include the mesh and hypercube. These configurations were chosen in part because real multiprocessors have been built with these configurations. The topologies in these models sometimes impose higher lower bounds on the time cost of problem solutions, due to the cost of communication over the network. Nevertheless, they often contain unrealistic assumptions. Such work sometimes treats the cost of transmitting a data item over a link as equivalent to the cost of computing a single value. For example, in [36], the authors make this assumption.

Early mesh and hypercube models shared with the PRAM model the unrealistic assumption that there is one piece of data per processor. As real parallel machines have been built, more sophisticated models have taken into account the fact that the number of processors is generally small compared to the size of the problems being solved. If P is the number of processors applied to a given problem and n is the number of data items to be processed in the solution of the problem, it is generally the case that

$$P \ll n$$

Sometimes the assumption that problem sizes are large compared to the number of processors, relaxes lower bounds on the time cost of parallel algorithms that arise due to the topology of the communications network. (See [17] for some examples).

These more realistic models include the Bulk Synchronous Parallel (BSP) model

of Valiant as introduced in [49]. The BSP approach treats the topology of the connection network as unimportant as long as the inherent parallelism of the program, which Valiant refers to as *parallel slackness* is somewhat greater than the actual parallelism available. The BSP model divides the computation into a series of phases or *super-steps*. Each phase consists of a computation stage and a communication stage. Particular parallel computers are characterized by the time it takes to perform the most costly communication step.

Practical research into the BSP model has shown that in some cases its predictions agree with actual performance. For example see [46].

The algorithms investigated here are not BSP algorithms. Instead they are chosen because the cost of communication is expected to be relatively small as problem sizes become realistically large. In such cases the cost of communication is not expected to impose insuperable obstacles to realizing useful speedups from the application of parallel techniques. The results in Chapter 6 show that this expectation is correct in the case of the algorithms implemented in this work.

2.1.4 Strategies for Parallel Solution of Computational Problems

When beginning to consider how to solve a problem in parallel, there are some overall design decisions that must be made. The choice made in a particular case will depend on the nature of the problem.

One broad distinction is between data parallel and functional parallel approaches. In a data parallel program, each of the processors runs the same algorithm, but each processor gets a different segment of a partition of the input data. Each processor works on its own subset of the data and produces a result which represents the solution to the problem for that part of the data. Generally, the results are then brought together in a single location to produce the solution to the global problem, but this depends on what is to be done next with the data. If the data represents an input to some subsequent operation, which is also a data parallel process, it may be convenient to maintain the result of the parallel computation in a distributed fashion.

With functional parallelism, each processor handles a stage of the work. The tasks done by each processor may not be the same. The outputs of one processor may be inputs to one or more others. Functional parallelism may be further subdivided according to the strategy for dividing up the work. One pure example of functional parallelism is pipelining, which is briefly described here for the purposes of illustration.

In pipelining, each processor takes on a different stage of a multistage calculation. The first element of input data passes to the first processor, which passes the results of its operation to the second processor in some fashion. The second processor will in turn pass its results to the next one and so on with each stage. While the production of a single result is a sequential process which will take at least as long as it would take on a single processor, the effectiveness of pipelining depends on the need to compute many results. As soon as the first processor has finished with the first piece of input data, it moves on to the next, passing it in turn on to the second processor. If the number of processors is P , and assuming that each stage takes the same amount of time, then after P stages, the pipeline will be full and all P stages of the computation will be performed at once for as long as new input data are available. After the pipeline has been filled a new result will be produced from the final stage of the pipeline at each step.

In this thesis we describe a data parallel implementation without dependencies among the processes (see Section 2.4.1), and a functional parallelization which also has no dependencies (see Section 2.4.3). The third program is a more complex, data parallel solution. It initially divides the input data among the parallel processes. The processes periodically exchange data in the course of the computation due to dependencies among them (see Section 3.3).

The first data parallel implementation is *parallel CanBEIS*. It is an ideal example, since the spatial information is partitioned once at the beginning of the process and the parallel processes are able to work independently thereafter.

The functional parallelization is of the Clarke Fire Model. The stages of the computation are divided among the parallel processors but the implementation is greatly simplified by the fact that all these stages of the computation are the same. The inputs to each of these stages are also the same. The only difference between

the runs on the several processors comes from the fact that during the course of the computation, some random values are generated that vary from one iteration to the next. This is the simplest imaginable form of functional parallelism. In this case, each processor performs a subset of the repetitive runs and the results are combined. In fact this form of functional parallelism bears a close resemblance to data parallelism except that, the input data are not partitioned, but instead each process starts with the full complement of input data.

Alternatively, if the randomly generated values are themselves thought of as inputs, then this can be thought of as yet another example of data parallelism, in which almost all input data are the same, but a small amount varies and that small amount is partitioned among the processors in the usual way. But it is more useful to think of it as functional parallelism, since the functional stages of the Clarke Fire Model program, (the various iterations) are partitioned while the input data are not.

2.2 Geographical Information Systems

In [28] a Spatial Information System is defined as *a computerized system for gathering, storing, presenting, manipulating, and interpreting spatial data for phenomena below, on or above the earth's surface. When the data are geographical it is called a Geographical Information System (GIS). Geographical means dealing with the surface (or the space close to the surface) of the earth. GIS data are present in vector or raster form.*

By vector form it is meant that the locations of data are represented explicitly in the model by their coordinates in some frame of reference. By contrast, a raster is a uniform grid which is imposed on the area of interest. The grid is built up by a division into rows and columns at regular intervals. The intersection of such a row and such a column is referred to as a *cell*. The physical area corresponding to a cell is determined by the size of the intervals which determine the rows and columns. If each cell has a width w_c and a height h_c , then the area associated with a cell has square magnitude $h_c * w_c$. The raster has a height h_r and a width w_r measured in cells. A raster row will have size w_r and a raster column will have size h_r . A raster

with h_r rows and w_r columns will thus have $h_r * w_r$ cells.

In a raster, there is a datum or data associated with each cell. All cells have the same number of bytes associated with them. Thus the cell to which a particular set of data is associated can be determined implicitly from its position in the block of raster data. The information needed to make this determination is the raster width w_r and raster height h_r as well as the order in which raster data appears. Commonly data is mapped row by row from the bottom row up to the top. This provides a compact way of expressing the spatial relationship between a set of data and the associated cells.

Representing data in vector form may be more efficient if data are distributed sparsely over the region of interest so that in a raster representation, most raster cells would contain useless data. Data for GIS are represented in both vector and raster form, whatever is more suitable for a given problem.

Geographical Information Systems (GIS) are of increasing economic importance due to their application to such large scale public management problems as forest fires ([14] and [25]), air pollution ([7]), and ice tracking, to name three areas where the Carleton GIS group is doing work. There are many other applications in business and government activities such as

- management of distributed physical plant by telecom and power companies
- routing problems such as traffic management and snow removal planning
- weather forecasting
- mineral exploration

and many others.

In [29] it is explained that *In recent years the demand on Spatial Information Systems technology from application areas has sharply increased. In particular, with the EOS (Earth Observation System), terabytes of spatial information will be collected daily. The increase in speed in sequential computing cannot keep up with the demand generated by many systems handling spatial data. Therefore a need for parallel computing in this area is widely recognized.*

It is the goal of the work described in this thesis to show the practical application of parallel techniques to GIS problems and further to demonstrate that significant improvements in performance can be obtained as a result.

2.3 Cellular Automata

Cellular automata (CA) were introduced by Codd [15] as an elegant mathematical model for a class of processes operating in discrete time and discrete space⁵. They are perhaps most generally known due to John Conway's Game of Life [22]. Cellular automata ideas have been applied to many fields including

- urban modeling (e.g., [41] and [4])
- international development (e.g., [18])
- medical modeling (e.g., [40])
- geological modeling (e.g., [26])
- ecological modeling (e.g., [10])

Here we are interested in particular in cellular automata in the context of GIS. CA have been recognized for some time as a powerful tool for GIS modeling and simulation. In [31], Itami discusses a GIS system built on a cellular automata foundation.

There are many examples of GIS modeling/simulation using cellular automata. Domains that have been studied include

- forest fires [25] and [14],
- forest infestation [27],
- earthquakes [6]
- forest growth and die-back [47],[34]

⁵[29]

- climatology [23]
- chaos theory [43]
- epidemiology [8] and [9]

Itami [32] has studied cellular automata for residential site selection (using Tomlin's Map Analysis package). His model was further refined by Gimblett, simulating the growth behaviour of urban and rural development in Indiana.

Brinch Hansen [25] describes a model program for parallel execution of cellular automata adapted to a (very simple) forest fire model and numerical solution of Laplace's equation for stationary heat flow. His intentions were to explore model programs for computational science, as he terms it, and cellular automata are one of several examples. We have adapted a variation on his simple fire model as a test program for the cell driver (see Chapter 4).

Bonfatti et. al. [10] describe the use of a cellular automaton to model the tidal flow in the lagoon at Venice, Italy.

In [48], Tobler introduces the notion of *Cellular Geography*; he classifies different cellular automata models covering a wide range of applications and generalizations.

More formally, he introduces the models described below, some of which are specializations of others. Our implementation is designed to cover these models as special cases. (See also the comments given below.)

1. Model I

The *independent model*: $g_{i,j}(t + \delta)$ is a random variable not related to $g_{i,j}(t)$. This model appears to be of limited use in the context of GIS.

2. Model II

The *functional dependent model*: $g_{i,j}(t + \delta) = F(g_{i,j}(t))$, where F is a function. The value of a cell at time $(t + \delta)$ depends only on the value of the cell at time t . This model assumes total locality.

3. Model III

The *historic model*: $g_{i,j}(t + \delta) = F(g_{i,j}(t), g_{i,j}(t - \delta), g_{i,j}(t - 2\delta), \dots, g_{i,j}(t - k\delta))$, where F is a function. The value of a cell depends on several previous values at that location. For example, if $\delta = 1$ then the value of cell $g_{i,j}$ at time $t + 1$ depends on the values the cell had during the last k time units. The model is based on local information only but takes into account the history of the cell.

4. Model IV

The *multivariate model*: $g_{i,j}(t + \delta) = F(u_{i,j}(t), v_{i,j}(t), w_{i,j}(t), \dots, z_{i,j}(t))$ where F is a function based on several attributes associated with the cell $g_{i,j}$ at time t .

5. Model V

The *geographical model*: $g_{i,j}(t + \delta) = F(g_{i,j}(t), \text{Neighbourhood})$ where F is a function defined for cell $g_{i,j}$ at time t using attribute values present at all cells in a neighbourhood of $g_{i,j}$. In its most general form, the neighbourhood is arbitrarily defined. Typical examples of some regular neighbourhoods are: squares around cells, the 4-neighbourhood (Von Neumann neighbourhood) and the 8-neighbourhood (Moore neighbourhood). See Figure 3.3.

It is this scheme that motivated the design of the Cell Driver (see Section 2.5). We endeavored to make it capable of supporting all of the above models of cellular automata problems.

2.4 The Applications

2.4.1 CanBEIS

The first GIS program which we ported to the Alex AVX II parallel computer was a portion of the Canadian Biogenic Emissions Inventory System (CanBEIS). This was our first study of the use of parallelism for real GIS applications and the origin of Carleton's parallel GIS project headed by J.-R. Sack.

CanBEIS is a program that computes the emissions of hydrocarbon gases that are produced as a result of organic processes. The gases are

- isoprene
- α -pinene
- monoterpene
- unidentified hydrocarbons.

These emissions are of interest because of their environmental effects. The inputs to the model are:

- surface wind speed
- ambient air temperature
- relative humidity
- opaque sky cover
- latitude
- longitude
- time.

Along with these environmental factors, the emissions are determined by the type of ground cover that exists in the area. For each type of land use, there is a particular pattern of emissions for the four gases, given the inputs. The CanBEIS developers obtained land use data for southern Ontario. They obtained records from weather stations from across Canada.

A Voronoi diagram was generated associating with each grid point the land area that is closer to it than to any other grid point. For each of these regions the CanBEIS developers computed the land use percentages. These percentages were then associated with the grid points in a raster using SPANS⁶, a commercial GIS package.

⁶SPANS is a product of Tydac Inc.

While the climatological data are only actually known at particular locations (weather stations), CanBEIS estimates the emissions for whole regions by interpolating the values of nearby weather stations at the points of a regular grid that is superimposed on the region of interest.

This may all be considered preprocessing. The portion of CanBEIS that was ported to the parallel machine performs the following steps.

1. Identify up to five Voronoi neighbouring weather stations for each grid point
2. For each grid point
 - (a) For each hour
 - i. Interpolate the meteorological data for this hour from the weather stations associated with the grid point
 - ii. Compute the emissions using the biogenic emissions inventory algorithm
3. Write out a table of total and maximum emissions with entries for each grid point

After the steps just enumerated SPANS was used to generate maps from the tabular data.

The portion of CanBEIS that actually computes the emissions was chosen as a candidate for parallel computing because it involves a great deal of computation, and because there seems to be a straightforward data parallel strategy available to implement a parallel version. A full description of the CanBEIS program is found in [7]. A discussion of the design of the parallel version appears in Section 2.4.1.

The sequential version of CanBEIS was developed by Wolfgang Bitterlich and Dominique Ratté for the Department of Energy Mines and Resources of the Government of Canada. He adapted it from a program called PC-BEIS that was developed using to compute emissions for any county in the United States by the US Department of Energy.

The outputs include total and maximum emissions of the four biogenic gases for the region associated with each grid point. These outputs are written back to files.

Computation using the original CanBEIS program of the emissions for a year on 278 grid points took 18 hours on an Intel 80486 processor running in a 486 Personal Computer. As the CanBEIS program is used frequently, it is important to reduce this time significantly using parallel computing.

One of the factors that makes CanBEIS a good candidate for parallelization is that the emissions at each grid point are calculated independently of the computation at any other grid point. This means that the input data can be partitioned among the processors to produce a data parallel solution to the problem (see Section 2.1.4 for an introduction to data parallelism).

With the data parallel approach, the results for the grid points associated with each processor can be calculated simultaneously, without the computation on one processor depending on the work being done anywhere else.

The independence of the computations at each grid point means that the algorithm used to compute the emissions on a sequential machine can be used without modification by a parallel machine. Each processor can solve the same problem on a small subset of the grid points using a small subset of the input meteorological data. Only when results for the whole map are to be displayed must the results from each processor be gathered together in one place.

For the purposes of our work, we reduced the scale of the problem to 46 grid points and 31 days to shorten the time needed for testing. The original CanBEIS program took 15 minutes to solve the reduced problem on a 486.

2.4.2 Overhead of Parallel Computation in CanBEIS

To parallelize the chosen portion of CanBEIS, the opening of data files in the original program had to be replaced with a procedure that opens these files and then distributes the data to those parallel processors that would do the calculation of emissions. The process which does the data distribution has to determine the partitioning of the input data, and hence of the work, among the available processors.

Pattern of Data Movement Parallel CANBEIS

Figure 2.2: Data Movement on a 16 Node Square Mesh of Processors

The output of the results to file has, similarly to be replaced with a process that gathers the results of the various subproblems from the parallel processors and combines them into an output that is similar to that of the original program. Since we wanted to display intermediate results, there is additional output introduced in our version. Figure 2.2 shows the processor network of a 2-d mesh and one possible pattern of data movement in this topology.

We expected that the computation time at each processor for the emissions would be much shorter than for the sequential program due to the fact that each processor is solving a small subproblem. On the other hand overhead due to communication is introduced when data is distributed to the processors as well as when results are returned from the processors to the host. The interesting question is: *Under what conditions are the additional costs introduced by the transmission of data to the nodes and the gathering of results afterward small enough, so that the improved performance promised by the parallel machine is in fact obtained?*

For a given problem size, the benefit available from parallelization can only go so far. There are at least two reasons for this. The first is that at some point we arrive at *computational atoms*, operations which are elementary in the sense that no improvement in performance would be achieved by dividing the work over more than

one processor even if instantaneous communication of data were available. In such an atom, each step depends on the output of the previous step. Such a computation is inherently sequential.

Before we reach the level of division into atoms, there is a coarser level of division which is likely to limit the degree of parallelism we can achieve and hence the number of processors that can be usefully employed. This second limitation is due to the fact that communication in a parallel machine is not instantaneous.

The communication cost for a data parallel program can be divided into setup costs and processing costs. The setup cost consists of loading the programs and input data into all of the parallel processors as well as of getting the final results out of the machine into some form that is useful for the end user. The processing costs arise due to the exchange of intermediate results between processors in the course of the computation. This communication is necessary when results produced on one processor are needed by others. The communication moves these results to where they are needed in order to continue the work.

The setup costs may grow in proportion to the size of the problem or they may be largely fixed. If they are fixed, then increasing problem size will decrease their relative significance and should lead to better parallel performance. Communication in the course of processing may grow in proportion to growing problem size, in which case increasing problem size may not improve performance. But if communication in the course of processing grows more slowly than the cost of computation, then growing problem size should lead to an improvement in performance.

It is the case that the cost of sending a relatively small message contains a high fixed setup cost, so that the exact amount of data being exchanged is not that significant. On the other hand, sending large amounts of data in the AVX yields a cost that is roughly proportional to message size. Therefore an understanding of how message costs scale with increasing problem size can give a clue in advance as to how a particular application will perform when it is scaled up.

In CanBEIS, the amount of data loaded into each processor is proportional to the number of grid points, but it occurs only once. Because a very lengthy computation is then performed on the data, this cost is relatively insignificant. On the other hand

the results are returned from the processors to the host repeatedly in the course of the computation. As the number of grid points on a node decreases, relatively small messages are sent frequently. As the number of grid points grows, relatively large messages are sent infrequently. Because communication is more efficient at sending large messages than small ones, we conjectured that the speedup for a given problem size would level off at some number of parallel processors and this is what we investigated.

The design of parallel CanBEIS is discussed in detail in Section 3.1 and the performance results are found in Section 6.2.

2.4.3 Clarke Fire Model

A second GIS problem studied is the spread of fires. Forest fires are of great economic importance, and a better understanding of their behaviour could be of great value. In this thesis, we describe the parallelization of an existing realistic forest fire model as a stand-alone program as well as the development of a highly idealized fire model in the context of the cell driver. The idealized model is based on one described by Brinch Hansen in [25]. We will introduce the idealized model in the next section. In this section, we introduce the Clarke Model.

The Fire Model program is a realistic forest fire modeling program developed by Dr. Keith Clarke of the City University of New York. The Clarke Fire Model is another example of a program that can be parallelized with relative ease. The Fire Model program uses a detailed model of fire propagation and applies it to data derived from satellite imagery of a region near Santa Cruz that experienced a severe fire.

Inputs to the model include a map of the fuel density in the region under study, a separate raster describing the fuel moisture, and a third raster which is used to characterize the topography. In addition, the model randomly generates wind direction and speed for each iteration, in order to test the input fuel and moisture data under a full range of possible wind conditions.

In a run of the program the random outbreak of numerous small fires is simulated and the model is run until all fires burn out. This is done repeatedly for a user-chosen

number of independent runs. For each run, the degree to which each cell has been burned is recorded. The output of the entire process is a raster in which the value associated with each cell is the probability that it will be burned.

2.4.4 Strategy for Fire Model Parallelization

The Fire Model program was originally investigated as a candidate for parallelization using the Cell Driver (see Section 3.3). However its strategy of executing multiple independent runs of the same algorithm on the same input data lends itself to a straightforward functional parallelization. As discussed in Section 2.1.4, functional parallelism involves the partitioning of the *operations* of the program among the processors.

Unlike CanBEIS, the data are not partitioned among the nodes. Instead, all data is sent to every node and the many trials that must be run on the data are divided among the processors. This approach is the simplest possible example of functional parallelism, since there are no dependencies among the various trials that are run until the data are combined at the end.

As with CanBEIS, this leads to a fairly straightforward design for parallelization. No communication is required, except at the beginning and the end of the computation. As a result we conjectured that a good improvement of performance could be attained.

An interesting feature of this problem was that a considerable amount of data had to be moved to the processors of the AVX II at the beginning and back from them at the end. A total of three 650 kilobyte rasters had to be loaded on to each AVX II node involved in the computation. At the end a 650 k raster has to be returned from each AVX node to the host program. In the case of a 64 node computation, a total of approximately 40 megabytes must be moved at this stage.

The parallelization of the Clarke Fire Model is described in more detail in Section 3.2. The results are described in Section 6.3.

In the next section we discuss the cell driver, together with the idealized fire model application that was developed on it. The cell driver is designed to support

data parallel cellular automata applications.

2.5 The Cell Driver

2.5.1 Introduction to the Cell Driver

The second fire model that we implemented is based on one described by by Per Brinch-Hansen in [25]. It was implemented in the context of the *cell driver* which we introduce in this section. Unlike the Clarke Fire Model, the Brinch Hansen program is a rather idealized model. It exists for the purpose of illustrating the potential of Cellular Automata to model natural phenomena, rather than to model fires realistically. As such it was a useful algorithm to test the cell driver, which is our own parallel cellular automata platform.

The cell driver (CD) is designed to facilitate the parallel solution of a large class of spatial problems that can be treated by the technique of cellular automata. (Section 2.3 contains an introduction to this class of problems). By focusing on the parallel aspects of the solution of cellular automata problems, the aim of the CD is to allow the development of parallel cellular automata applications with the same ease as the development of such applications for a conventional sequential machine.

This is an ambitious goal, due to the great variety of cellular automata models (as summarized in Section 2.3). However the achievement of the goal in many cases is aided by the fact that a large subset of cellular automata applications fall into one or two subclasses from the point of view of the parallel issues. While the open design of the cell driver does not rule out any of the models discussed in Section 2.3, the development effort invested thus far has focussed on implementing the parallel behaviour of these subclasses.

The CD was developed for the parallel platform because cellular automata problems as a class seem like good candidates for parallelization. While the subproblems are not independent as they are in the first two applications discussed in this chapter, there is reason to believe that the ratio of communication to computation will be sufficiently low for realistic cellular automata problems. The reasons for this are

discussed in Section 2.5.3. First we put the Cell Driver in its context which is the Neighbourhood Modeling System (NEMO).

2.5.2 Neighbourhood Modeling (NEMO) System

The Cell Driver is a component of the Neighbourhood Modeling (NEMO) System. NEMO has been the principal project of the PARAllel DIGItal Modelling (PARADIGM) group, as the Carleton GIS research group is called. The goal of this large project is to provide parallel support for the development of *raster* based GIS applications. Besides the author of this thesis, three doctoral students, (David Hutchinson, Mark Lanthier and Doron Nussbaum) two other masters student (Nashib Qadri and Zhengping Wang), an international visitor (Lars Küttner of Humboldt University), and several coop students have worked on NEMO components and applications over the course of a two year period. The entire project is over-seen by Prof. J.-R. Sack.

NEMO is funded in part by Almerco Inc. and by NSERC. NEMO forms the basis for a research program on parallel spatial modeling for which Carleton University has been awarded a research chair position, with chair-holder J.-R. Sack.

As first discussed in Section 2.2, a *raster* is an implicit array of data in which two dimensional spatial data are represented by a block of data together with a cell size, a block height and a block width. The cell size, block height and block width provide the key to associating each cell in the block with its correct location in space.

In addition to the cell driver, NEMO contains the Neighbourhood Analysis (NAN) driver which is designed to support the parallel computation of spatial statistics on very large rasters; and the propagation driver (PD) which supports the modeling in parallel of processes in a raster which spread along an active boundary.

The NAN driver is currently being used to support the development of an implementation of the Tomlin Map Algebra. The propagation driver is being used to support the development of another realistic fire model due to Mike Wotton at the Canadian Department of Forestry (see [24]) as well as a terrain skeletonization algorithm (see [2]). Both of these projects are being developed by Nashib Qadri.

Together with the drivers, NEMO consists of a *raster data base manager* (*raster*

manager), a *display manager* and a *user interface*. The raster manager takes care of the management of input and output of data. This is an important function, since raster GIS applications typically operate on very large data sets. It is necessary to manage the flow of data so as to use the communication bandwidth available in the AVX efficiently. The raster manager can also incorporate data compression techniques to speed up communication.

The display manager makes it possible for any NEMO application to produce a graphical output of a raster image. It handles the display of raster components from the distributed processors of the AVX. The components can all be displayed in a common window. The availability of the raster manager and the display manager tools relieves the developers of the drivers from the need to implement I/O, and allows them to focus on the solution of the particular class of problems that is the province of a driver.

The drivers, in turn, are intended to allow application programmers to focus on the specific problem to be solved, while the driver handles the issues that arise because the program is being implemented on a parallel platform.

The NEMO user interface program provides a menu driven access to the various applications implemented using the NEMO drivers. Applications are available under menu headings labeled with the various drivers. The user interface also allows the user to choose the size of mesh and hence the number of processors that the application is to run on.

The cell driver is designed to work with these NEMO components, getting input from the user interface, displaying intermediate and final results using the display manager and opening and saving rasters using the raster data base manager. In addition to the Brinch Hansen fire model which is discussed here, another application currently being implemented with the cell driver is an earthquake modeling system. This work is being done by Zhengping Wang.

The NEMO project is summarized in [29] presented at the Seminar on Parallel Algorithms and Applications (SPAA) in June of 1996 and described in greater detail in [30]. The Brinch Hansen application is described in Section 2.5.4.

2.5.3 Criteria for Beneficial Parallel Implementations of Cellular Automata

The cell driver is intended to manage the complexity of programs that require some interchange of intermediate results between the processors to perform their function. In particular, for CA problems that fit the *geographical model* (see Section 2.3) to update the value of a given cell c of a raster, the value associated with a number of other cells (called the *neighbourhood* $N(c)$) must be known. If we take a data parallel approach and divide the initial raster up among a number of processors, it will be necessary for processors holding neighbouring pieces of the raster to exchange data in order to perform the required computation.

Cellular automata work by running many iterations of the same computation. At each iteration the value associated with each cell in the raster may change. In particular, the values of the cells in the neighbourhood may change. Therefore it is necessary for data exchange to occur at each iteration between processors holding neighbouring pieces of the raster. Thus, unlike CanBEIS and Fire Model, the parallel processes that are developed to implement cellular automata applications are not independent. Achieving desired speedups in this situation is more difficult than it is for computations such as CanBEIS and the Fire Model where data transfer and process synchronization must occur only at the end of the computation. Whether the desired speedup can be achieved depends on two factors:

1. Communication overhead
2. The balance of work among the processors.

Communication overhead introduced by the exchange of data among the processors can be minimized if the amount of data exchanged is kept small and if the distance the data must travel is minimized. It is also important to avoid bottlenecks in the communication such as those that occur when all the processors attempt to communicate with one particular processor at the same time.

While there are a number of Cellular Automata models that update the value of a cell based solely on values associated with that cell (see Section 2.2), cellular

automata problems typically involve the processing of each cell in a raster by looking at the values of the neighbourhood $N(c)$ of the cell. If the neighbourhood is small relative to the piece of the raster assigned to a processor, the amount of data to be exchanged will be small relative to the total size of the problem. Under these conditions we conjectured that the communication overhead would be small compared to the computation cost of each subproblem and that as a result a good speedup would be obtained from the application of parallel processing.

In order to gain benefit from parallelism, it is also necessary that the computational work be distributed as evenly as possible among the processors (*load balancing*). This is because the length of time for the parallel computation as a whole must be at least as long as the time taken on any one of the participating processors.

For applications of cellular automata, a good balance can be achieved by dividing the raster in proportion to the expected work, assuming that the amount of work associated with the regions is known in advance. If this is not the case and if unbalanced loads are expected, dynamic reallocation of work may be required. However, this work focuses on those situations where a static division of the raster will suffice. This is clearly preferable if it can be achieved. It avoids the communication overhead that arises from dynamic load balancing and leads to greater simplicity for the resulting implementation.

The simplest load balancing scheme is one where a roughly equal amount of work is done on each cell of the raster at each iteration. The simple fire model application based on the work of Per Brinch Hansen [25] is an example of such a problem. The Brinch Hansen application is discussed in Section 2.5.4. Performance gains achieved for this application are presented in Section 6.4.

In cases where the work is proportional to the number of cells, the work load among the processors can be balanced by giving each processor an equal sized piece of the raster to process. It is problems of this kind that were investigated using the cell driver.

The cell driver is designed to support applications that solve many different types of cellular automata problems. Care was taken to avoid design decisions that would restrict the type of applications that can be developed on it. Thus many decisions are

left up to the application designer. For example, workload balancing issues will differ with each application. The division of a given problem among the available processors depends on the work that is associated with the various problem components.

However, the case where the amount of work per cell is roughly equal is a good candidate for beneficial parallelization. The balanced division of a block structured raster among the available processors is built into the driver as a default behaviour. If this division strategy is appropriate for a given application the developer is saved the effort of writing this cell driver component.

We have shown (see Section 6.4) that cellular automata applications that generate the result for each cell based on a small surrounding neighbourhood can benefit greatly from parallelization since the amount of computation is large relative to the amount of communication. Since this scenario is a good candidate for beneficial parallelization, the neighbour data exchange behaviour for such applications has also been incorporated into the driver. The built-in functions are useful for any application with a symmetric neighbourhood that exchanges data only with the eight nearest neighbours. Developers who can make use of this behaviour along with the default raster partitioning behaviour will find that most of the parallel issues involved in developing a parallel CA application have been dealt with for them by the cell driver.

As other useful scenarios are identified, these too can be incorporated into the driver, further facilitating the development of parallel CA applications.

2.5.4 The Simple Fire Model of Brinch Hansen

The fire model implemented as our initial cell driver application was proposed by Brinch Hansen in [25]. He cited it as an example in order to illustrate the simplicity of the CA approach in modeling complex natural behaviour.

In it, each cell has one of three states, alive, burning or dead. The neighbourhood used in the model is the four nearest neighbours of each cell.

The rules of the cellular automaton are simple. It is of the same order of complexity as the *game of life* [22].

1. If a cell is in state *alive* then it has some probability at each iteration of changing

state to *burning*

2. If a cell is *alive* and has at least one neighbour that is *burning*, then at the next iteration the live cell changes state to *burning*.
3. If a cell is *burning* at one iteration then at the next iteration it's state is *dead*.
4. If a cell is *dead* then at each iteration it has some probability of becoming *alive*.

If we initialize the raster so that all cells are initially *alive*, this algorithm produces a very idealized model of fire propagation and forest regrowth.

In the application as implemented in the cell driver, the rules were modified to produce a slightly less predictable behaviour, but the essential features of the algorithm, the neighbourhood and hence the neighbour data exchange are the same as for the original algorithm.

The parallelization of the Brinch Hansen algorithm was expected to produce good results because the neighbourhood around each cell is small. As a result the amount of data to be exchanged between processors is expected to be small. Other than this data exchange, the computation on each processor is independent of that on the others.

The design of the simple fire model application is described in Section 4.4 and the parallel performance results are reported in Section 6.4

Chapter 3

Design Issues

3.1 CanBEIS

As first mentioned in Section 2.4.1 the overall approach used in design and development of the parallel version of CanBEIS is to take advantage of the independence of the tasks to be done at each grid point.

The input data is partitioned among the nodes. The weather station data is loaded into each node. In our demonstration application, our main interest was in performance and not in the geographic results. For this reason, the same three weather stations are used to interpolate data at every grid point. (In general, the computation time to determine the Voronoi neighbouring weather stations is small). The landscape data is loaded into every node as well. The land use data is a fixed quantity and is hard coded in the node program.

Once the input data has been loaded into the AVX, the original algorithm from the sequential CanBEIS can be run on each processor. Results are computed for each grid point and for each hour in the time over which the simulation is run. These results are sent back from the processors to the host program. In place of the file output, which was produced by the sequential version of CanBEIS, we substituted a graphical display. This display, developed by Doron Nussbaum, provides a general impression of the level of emissions associated with each grid point (see Figure 3.1) and, in a separate graph, the level of emissions associated with each hour of the day.

Figure 3.1: Graphical Display of Total Gas Emission in CanBEIS. Vertical axis represents total emissions while horizontal axes represent the distribution of the grid points over the area under study.

The display was added to allow us to visualize the progress of the parallel computation and to be able to view intermediate results. In order to do this we had to change the order in which the original algorithm computed results. The original algorithm computed the results for the entire time of the simulation for a given grid point before going on to the next one. In order to be able to display the cumulative results after a certain number of hours in the model, we had to change this so that the results for every hour were computed for each grid point before going on to the next hour.

The development of the parallel version of CanBEIS involved the movement of the software from the DOS environment to the Unix and Trollius environments. Certain idiosyncrasies of the C language on the PC platform led to the need for modifications. For example, integers on the PC are by default two bytes long while they are four bytes long in the Unix and Trollius environments. The PC version of CanBEIS used the fact that integers are two bytes long in a few places. These dependencies on the PC platform had to be removed.

Another issue that arose was the compatibility of data being transmitted across heterogeneous platforms. The host processor of the Alex AVX II is a Sun SuperSparc processor on a Sparc 20 board and the other processors in the AVX are T805 transputers and Intel 860 vector processors. The SuperSparc processor represents data differently than the other AVX processors and there is a need to convert the data from one representation to the other.

The Trollius operating system provides a means of conversion for simple types and for arrays of simple types, but not for structured types. We discovered that using the Trollius conversion functions on the fields of structured types does not necessarily guarantee that the structure will be transmitted correctly. Arranging the fields of the structured type so that larger fields precede smaller did seem to ensure a faithful transmission. This is the case because hardware aligns fields so as to put them on word boundaries if the boundary would otherwise fall in the middle of the field. If two hardware platforms behave differently the structure from one may be misread on the other. For example: Let a structure contain a character and a long integer. If one platform aligns fields on a four-byte word and another on a two byte word and the

character comes first, then the long would start in byte two on one machine but be expected to start on byte four on the other machine. Since Trollius does not correct for this the two structures will be wrong after they are transmitted. A strategy that resolves this problem is to put longer fields before shorter ones. We can see that in our example, if the long precedes the character, then both platforms would arrange the structure in the same way.

Familiar tools also could fail in the new environment. The conversion function for floating point numbers was used in our implementation and the portion of the program that was run on the host computer was compiled with the GNU C compiler gcc. However we discovered that for some reason the conversion function for floating point numbers did not work correctly when the program was compiled with gcc, though it worked fine when the Sun C compiler cc was used instead.

We also discovered "bugs" in the operating software that was supplied with the AVX II. In particular, the floating point computation using the math library did not work on the T805 transputer, although it worked correctly on the Intel 860. Another problem was that pointer bugs in the message passing calls in Trollius do not just result in the abortion of the program but also bring down the Trollius operating system itself. Once Trollius ceases to run on a node, message traffic through that node is no longer possible. Normally this will bring the entire parallel program to a halt without any indication of the cause of the problem. This can lead to some challenging debugging situations.

Finally, there were also occasions when certain processors did not run the program even though it ran correctly on others. Sometimes we found that the cause of these problems was a fault in the hardware. Distinguishing between the hardware bugs, operating system bugs and the bugs in our own software made the development of parallel CanBEIS a sometimes difficult challenge. This was a motivation for the development of the eXtended AVX Simulator (XAS) (see Chapter 5) which, among other benefits, provides a reliable simulated hardware platform on which to run Trollius programs.

The work on CanBEIS led to insight into the sort of tools that it would make development on the AVX easier. Other than XAS, a message passing performance

visualization tool, *vis* was developed by Craig MacDonald, using the parallel CanBEIS program for testing.

The results of timing performance of parallel CanBEIS on different numbers of nodes appear in section 6.2.

3.2 Fire Model

The design of the parallel version of the Clarke Fire Model drew successfully on the experience with CanBEIS. As with CanBEIS, data are loaded into the machine at the beginning of the process and results are then returned by the processors as they are computed to a host program that combines them into a final result.

However, there are some important differences between the Fire Model and CanBEIS as well and these differences led to different decisions on the design. As a data parallel program, parallel CanBEIS loaded different data into each processor. Since each processor received different input data, a separate message had to be sent from the host program to each of the node programs.

In the Fire Model, all the input data to each processor are the same. As a result we are able to take advantage of the multi-casting facility that is built into the Trollius operating system of the Alex AVX II. Broadcasting saves on communication by taking a single copy of the broadcast message from the sender and copying it as needed to distribute it to all the nodes. If we instead sent a separate message to each node, the same message would travel over some of the connections in the machine many times.

Unlike many functional parallel programs, there are no dependencies between the different parallel processes. They are thus all free to run to completion as quickly as possible without ever having to stop to wait for input from any of their neighbours. Also, unlike parallel CanBEIS, results are only sent out from each processor once at the end of the computation.

One of the challenges of Parallel CanBEIS program was the comparatively large amount of data that had to be moved into and out of the AVX II. As we see have shown in Section 6.2 this data flow limits the speedup achievable for a given problem size. With the Fire Model program an issue arose because we used the built-in

network message passing software of the Trollius operating system to move very large individual messages.

The test data provided with the Fire Model program consist of a number of 620 row by 526 column rasters, with a short integer associated with each of the 326,120 cells. The output consists of another such raster representing the probability of each cell being burned. Since each short integer occupies two bytes, each of these rasters consists of 652,240 bytes.

In the parallel version, we initialize the node processes by sending each of the input rasters from the host to the other nodes. This is accomplished using the Trollius multicasting facility to broadcast a single network message with a 652,240 byte message body. In early tests of the parallel version of the program, these messages were not received. Checking the machine with one of the AVX diagnostic tools revealed that parts of the initial message were being buffered within the machine, but none was being delivered.

Moving the Fire Model to XAS, revealed the problem. There are thresholds associated with the Trollius buffering system on each node, beyond which the system will no longer accept user messages. XAS prints out a diagnostic message when these levels are reached, while the AVX does not. Once these levels are reached, user or system processes attempting to send a packet to the buffering system will block until more space becomes available. If no process is able to receive any of the data stored in the buffering system the result is deadlock. We discovered that the default level for these thresholds was smaller than the size of one of our input messages. Since the system will not deliver a multi-packet message until all packets are present on the local node, these messages could never be delivered.

It is odd that the default levels for these thresholds are so low, since each AVX node has 16 megabytes of dedicated Random Access Memory that is used only by the operating system, principally for message passing. Fortunately these levels are easily reset by a command issued from the command line in the actual AVX. Once they were reset to 4,000,000 bytes, the data broadcast from the host program were received on the nodes without further problem.

We also attempted to gather the results from the nodes to the host using the

Trollius multi-reeling feature. However, the multi-reeling seems to hang up with the large messages used in the Fire Model. The cause of this problem is as yet not understood.

In order to resolve this difficulty we resorted to the transport layer functions *tsend* and *trecv*¹ which send each result raster out to the host after establishing a connection. As a result the output is sent to the host sequentially as each processor takes its turn. For a 64 processor computation, this process takes about two minutes. This is the principle overhead introduced by parallelization.

In addition to the speedup achieved by parallelization, we were also able to significantly improve the time performance of the sequential version of the Fire Model program. This was done at the cost of using an additional 656 kilobytes of space. The parallel version of the Fire Model program that was implemented is based on this modified version of the sequential program.

The sequential program received from Dr. Clarke conserves space by reloading the fuel data input raster at each iteration. The fuel data are modified in the course of the computation and as a result must be refreshed at the beginning of each iteration. The original version of the Fire Model does this by reloading this raster from disk. The equivalent parallel action is to reload this raster at every iteration from the host to the nodes. This produces a relatively poor performance on the part of the parallel version, due to the very large amount of data being moved into the AVX machine (64 copies of a 656000 byte raster or approximately forty-two megabytes at each iteration).

We modified the original version of the Fire Model by adding an additional 656 K raster to cache the initial value of the fuel raster. Using this extra space, we replaced the loading of the fuel data from disk with the copying of the original data into the fuel raster at the beginning of each iteration. This change improved the performance of the sequential version of the program by about 32%, before we added parallelism. The elimination of the reloading at each iteration also greatly improved the performance of the parallel version. Section 6.3 describes the performance results of the parallel Clarke Fire Model.

¹The transport layer is introduced in Section 2.1.2

3.3 The Cell Driver

Some of the parallel design issues for cellular automata were introduced in Section 2.5.3. Here we focus in more detail on two of them. These are the partition of data among the processors and the exchange of neighbour data between processors holding neighbouring pieces of the raster at each iteration.

3.3.1 Partitioning of Data Among the Processors

In order for any application to achieve the benefits of parallelization the work must be divided among the processors in a balanced fashion. For a cellular automata application, the work can depend in a complicated way on the state of the raster. However, if it is the case that the amount of work to be done on a section of raster at each iteration is roughly proportional to the number of cells in that portion it will be possible to achieve load balancing by dividing the raster as evenly as possible among the processors.

Getting the data to the processors where it is needed is a matter that can be coordinated by a managing process, generally running on the host. This processor can read data and send it to the nodes itself, or it may simply determine a division of the raster and let each node process retrieve the data it needs from a central source. Alternatively each node process may independently calculate the data partition. If each of them runs the same algorithm then all will be in possession of the global map.

Because we are partitioning a rectangular raster among processors that will have to exchange data with other processors holding neighbouring pieces of the raster, it is natural that we connect the processors into a *two dimensional mesh*. This provides a natural mapping from the raster to the processors. It is also of great benefit if the processors holding neighbouring pieces of data are themselves neighbours in the parallel machine. This will minimize the distance that neighbour data must travel between processors when it is exchanged. The two dimensional mesh is the right topology for this application and is assumed to be used throughout this discussion.

If the raster has a block structure, and the number of processors in a mesh row and column is arbitrary, the design of a general algorithm is a task of some complexity.

The pieces of raster allocated to each processor may not generally be exactly the same size. There is a need for a general algorithm that allocates blocks to processors in as balanced a way as possible. The provision of this service is one of the ways in which the cell driver can lower the threshold of difficulty to the creation of parallel cellular automata applications.

The rasters maintained by the raster manager have a definite block structure. These blocks have height and width equal to a power of two. For example the blocks of the raster used for the Brinch Hansen application are deemed to be of size 16 by 16 cells. These blocks are indivisible units as far as the raster manager is concerned. They can be retrieved altogether or not at all. Thus for convenience and algorithmic simplicity, the block structure of the raster is allowed to determine the boundaries of the units allocated to a particular processor.

Another issue is to keep the logic of the neighbour data exchange functions manageable. This imposes another restriction on the data partition. The elements of the partition must be *rectangular*. Given these constraints, the partition of a raster in which the amount of work associated with each cell is roughly equal can be achieved by the following algorithm.

Let b_w be the number of blocks across the raster and let b_h be the number of blocks running vertically up the side. Let p_w be the number of processors along a row of the mesh of processors and let p_h be the number of processors running vertically up a column of the mesh. Then the minimum number of horizontal blocks allocated to a processor will be b_w/p_w and the minimum number of vertical blocks allocated to a processor will be b_h/p_h . If p_w divides b_w this will be the number of blocks across allocated to every processor. A similar argument applies to the vertical dimension.

But in general p_w does not divide b_w and p_h does not divide b_h . In this case there will be $\text{mod}(b_w, p_w)$ processors with an extra block across and $\text{mod}(b_h, p_h)$ processors with an extra block in the vertical dimension. For convenience, we let the first $\text{mod}(b_w, p_w)$ processors along the mesh row be the processors with the extra blocks and the first $\text{mod}(b_h, p_h)$ processors along the mesh column be the ones with the extra block vertically. In this case, the largest section of raster allocated to a process will be $(b_w/p_w) + 1$ by $b_h/p_h + 1$ blocks and the smallest will be (b_w/p_w) by b_h/p_h blocks.

Figure 3.2: Partitioning an Arbitrary Raster Over an Arbitrary Mesh

This scheme results in at most $b_w + b_h - 1$ block difference between the largest and smallest rectangular piece allocated to a processor, linear in the size of the perimeter of the raster block. The overall work which depends on the product of the height and width of a block will only differ by the sum of the height and width. In general this means that the work on all processors will be of the same order of magnitude.

In Figure 3.2 we see an example of the application of this partition algorithm. A 16 by 16 block raster is mapped to a 3 by 3 two dimensional mesh of processors. In this case b_h and b_w are both 16 and p_h and p_w are both 3. The integer ratio b_h/p_h is 5 (same for b_w/p_w) and $\text{mod}(b_h, p_h)$ and $\text{mod}(b_w, p_w)$ are 1.

So the first row and first column of the mesh have 6 rows and 6 columns of blocks associated with them respectively. The other rows and columns of the mesh have 5 rows and 5 columns associated with them respectively. The resulting partition of the raster is depicted in Figure 3.2.

3.3.2 Neighbour Data Exchange

While the range of cellular automata models discussed in Section 2.3 includes several that do not look at the neighbourhood, the preponderance of practical applications listed in Section 2.3 do so. They may look at cell history and cell attributes as well. But because the historical data can all be maintained locally and attribute data can be made available on every node at the start with no further communication required, neither historic nor multivariate models give rise to the sort of communication requirements that arise with the geographical model.

If the value for a cell at time $t+1$ depends on the values associated with some neighbourhood at time t , and the raster is partitioned among the processors, then at each iteration it will be necessary to exchange data with the processors that hold neighbouring raster pieces. The amount of data to be exchanged depends on the size and shape of the neighbourhood.

Some two dimensional cellular automata use only the four nearest neighbours (the Von Neumann neighbourhood) or the eight nearest neighbours, (the Moore neighbourhood). For example, the Brinch Hansen fire model algorithm (discussed in Section 2.5.4) uses the Von Neumann neighbourhood. In this case, only the four nearest neighbouring processors are involved in the data exchange. If the Moore neighbourhood is used, eight processors are involved.

More generally, if the neighbourhood involved in the transition function is cross-shaped, and centred on the affected cell, only four neighbouring processors will usually be involved in the exchange of data. Otherwise at least eight will be involved. Figure 3.3 shows examples of neighbourhoods that require data exchange with four and eight neighbour nodes.

Unless the neighbourhood is large in comparison with the size of the portion of the raster associated with each processor, the exchange of data should be limited to the eight nearest neighbouring processors. We expect that many cellular automata applications that can benefit from parallelization will fall into this category.

The algorithm for general neighbour data exchange even with a restriction to the eight nearest neighbours is non-trivial. This is another mechanism that would have to be implemented in any stand-alone parallel cellular automata application.

Data Exchange with 4 Neighbours **Data Exchange with 8 Neighbours**

Figure 3.3: Examples of Cell Neighbourhoods that Require Data Exchange with 4 and 8 Neighbours

Therefore, this service is built into the driver as an option available to be called by those developers that can use it. For an application which can use the default behaviour the result should be to greatly lower the degree of difficulty for development of parallel cellular automata applications.

The implementation of this optional behaviour is found in the *Default_Setup_Neighbour_Info*, *Default_Send_Neighbour_Data* and *Default_Get_Neighbour_Data* functions. The algorithms for these functions are found in Chapter 4.

Thus in spite of the great diversity of potential applications that could be put on the cell driver the driver itself can supply some very useful services. This helps us to achieve our goal of transparent parallelism for developers of cell driver applications.

3.3.3 GIS Issues

The data associated with a particular raster cell can be divided into three types. These are

1. Spatial Data
2. Temporal Data

3. Attribute Data

By *spatial data* we mean, the changeable properties associated with each cell as they are at the present moment. If the current time is t , then the **spatial data** are the values associated with each cell at time t .

By *temporal data* we mean the history of a cell. This may include all the values that were associated with the cell from some time $t - k$ in the past up to time t .

Attribute data is associated with each cell and affects the application of the rules of the cellular automaton, but does not itself change as a result of the application of the transition function.

The cell driver must have the capacity to store spatial, temporal and attribute data on each node so as to support all of the models defined in Section 2.3.

The resolution of the cell driver design issues raised here is discussed in some detail in Chapter 4.

Chapter 4

Cell Driver: Goals, Design and Development

4.1 Management of Generality

The cell driver was conceived as a platform that is sufficiently general to support any cellular automata application. Because cellular automata can be used to treat a great variety of different problems and because these problems can vary in the amount of temporal, spatial, and attribute data that they use, very few assumptions could be made about what behaviour should be provided to support every such application.

However, there are a series of phases that every cellular automaton implementation must go through. The driver was written to proceed through each of these phases in turn. Where the behaviour cannot be known in advance, a user written component is called by the driver. By writing these components, the user retains flexibility in the behaviour that will be exhibited by the particular CA application that is being developed. The user written components handle the parts of the cellular automaton that are application specific. They also set various flags in order to choose among the behaviours available through the driver.

In addition to the variability of behaviour, the cell driver needs to be able to handle arbitrary data structures. It must be able to accommodate an arbitrary number of rasters as inputs, to store an arbitrary number of previous generations, with an

unknown amount of attribute data. This is necessary in order to accommodate all of the models described in Section 2.3. Given these inputs, a user-defined transition function must be applied to them in order to produce an arbitrary result raster or rasters. These rasters may need to be displayed or saved, possibly at each iteration. Under these conditions the driver itself can only supply pointers to the various types of data that are known to be needed, leaving it up to the developer to supply the details.

In addition to limiting the amount of functionality we could build into the driver, this quest for generality also seemed at first to clash with one of the main goals of NEMO which is to hide the parallelism from the user. Such parallel issues as the initial division of the input data among the processors, and the exchange of data between the processors, depend on the application. It would seem then that the user must write a data partitioning function and a neighbour data exchange function, both of which take the parallelism into account. In particular, the neighbour data exchange functions require that the application developer write a message passing parallel application, a programming specialty which most programmers do not have. If this were generally to be the case, the benefits offered by the cell driver in easing the development effort required for parallel CA applications would be limited and the population of developers interested in building applications on it would be small.

However, as we have discussed in Section 3.3, in many cases the requirement that the developer write a data partitioning function and a neighbour data exchange function can be avoided in practice. This is accomplished by including, as defaults in the driver, functions that handle the most common data division and neighbour data exchange situations. The developer deciding whether or not to use these defaults must have some insight into the parallel issues, or the utility of the default behaviours for a given application can be determined experimentally. If it turns out that their use is appropriate, much coding effort can be avoided.

If the default behaviour is appropriate for an application that is being developed, the original coding by the developer may be largely limited to a transition function as would be the case for a sequential implementation. The transition function, which computes the next iteration of a raster given the previous one is discussed in more

detail below.

If the initial iteration of the raster exists in a file, the driver will open it. Otherwise the user must also write a function to create the initial iteration of the raster and initialize the data structures that would normally be initialized when the driver opens a raster. If the user wishes to display the results using the display manager, a function to convert the raster data to a form understood by the display manager will also be required.

The other functions which the user must write will call the default neighbour passing functions and set a few flags.

The default raster partitioning behaviour divides a raster in a balanced way among the processors, given only its total height and width, the number of bytes in a cell and the height and width of a raster block. This function was tested with the Brinch Hansen fire model, which is our test application for the cell driver (introduced in Section 2.5.4). For this application the default neighbour data exchange functions produced good results as shown in Section 6.4).

The Brinch Hansen application involved a simple transition function that does roughly the same amount of work per cell regardless of cell state. This means that the load can be balanced among the processors by dividing the raster as evenly as possible among them. This even division of the raster, taking into account the block structure of the application, enabled the use of the *Default_Compute_Map* function (see Section 3.3.1). This function can be used by any application that needs to divide a raster evenly among the processors, regardless of block structure and regardless of the number of processors.

The default neighbour passing behaviour works with the *Default_Compute_Map* function but will also work with any rectangular partition of the raster as long as the *Global_map* has been set up, and the raster header data has been initialized.

The size and shape of the neighbour data exchange buffers depends on the size and shape of the neighbourhood involved in the calculation of the new state of a cell. The buffer size also depends on the number of bytes in a cell. The problem is more complex if there is more than one raster with data that has to be exchanged. These issues also affect the behaviour of the neighbour data sending and receiving functions.

All of this information is available in the header structure returned when a raster is opened. This allows the driver to handle it transparently for the cases covered by the default behaviour. In the current implementation of the default behaviour, one raster is assumed.

See Section 4.2.4 for a discussion of the default neighbour passing behaviour. In the next Section (4.2), we discuss the components of the cell driver in more detail.

4.2 Components of the Cell Driver

4.2.1 Overview

The cell driver can be logically partitioned along several dimensions. The most coarse division is between the node and host components. The host component is responsible for reading the parameters, creating a raster file or files if necessary, opening a display window and launching the node processes.

The host process also synchronizes the display of intermediate results in the display window. The host program must receive notification from the nodes when they have displayed their component for a particular iteration using the *ras_display_image_and_wait* function of the display manager to write their component of the image to the back buffer. When the host program has heard from all the node programs it notifies the display manager with a *swap_buffers* call to swap the back and front buffers. This ensures that the image displayed contains components that all correspond to the same iteration and that the image is only displayed when all the node programs have supplied their portion of the data. The host program then acknowledges to all of the node programs so that they can display the next iteration.

The node programs are responsible for the rest of the work. They open the raster file or files if necessary, or create the initial raster iteration. They do an initial exchange of data with their neighbours. Then for the number of iterations requested by the user, they

1. run the transition function

2. display the results if necessary
3. save the results if necessary
4. set up for the next iteration.

In the case where data are displayed graphically, there is normally a conversion required from the format of the application to the format understood by the display manager. The driver calls a user-written function *Convert_to_Display_Image* to accomplish this. In the future, a function can be added to the cell driver that will take a map between the user's representation and the display manager's colour scheme and perform the conversion transparently.

The node and host components can in turn be decomposed into driver level and application level code. This partition is useful to developers trying to understand what the cell driver provides and what is required to be written.

4.2.2 Application and Driver Levels on the Host and Node

The driver level on both host and nodes contains a loop that runs the driver through the iterations of the cellular automaton program, calling user written functions interspersed with driver calls. On the host this loop is provided by the *cell_driver* function which the application level program calls on the host after setting up.

On the host, the application supplies the main program. It is therefore responsible for receiving the standard cell driver parameters from the user interface. (This is an operation that can be moved into the driver in the future). It can then open a dialogue box to get any application specific parameters.

The application level is also responsible for starting up the node programs. A function called *launch_processes* is provided as part of the cell driver Application Programming Interface (API) in order to facilitate this. The *launch_processes* function specifies the executable file to launch and the parameters to be passed to the process when it starts up.

After the node programs have been started the host application should call the *cell_driver* function. The *cell_driver* function

1. opens a display window
2. optionally computes and distributes the partition of the raster among the processors
3. calls the *create_raster* function
4. goes into a loop where it keeps track of messages from the node programs telling it when to display and when the work is done. Whenever all node programs have displayed, the host program will tell the display manager to swap buffers. When it has heard from all node programs that they are done, it cleans up and terminates.

The *create_raster* function must be written by the developer and is part of the host application level. This function may create a raster file using the raster data base manager interface so that the node programs can open this file to get their initial iteration for the run of the cellular automaton. If no file is needed this function just returns *CELL_OK* without doing anything.

On the nodes, the main program is supplied by the driver. It receives the command line parameters which are supplied by the host program when the node processes are launched. It then initializes the raster block for the processor where it is running, by opening the file created by the *create_raster* function on the host, if any. An alternative is to open some existing file. If no raster file is to be opened, the application initializes the raster data on the nodes as part of the *Initial_Setup*. In this case, if the user intends to use the default raster partitioning and neighbour passing functions, the raster header data is set up in the *Initial_Setup* function as it would be if a raster were opened. A function *setup_ras_header* will be added to the cell driver API which can be called from the user written *initial_setup* function.

In addition, the node program optionally computes the raster partition. The cost of transmitting the partition from the processor that computes it to the other processors can be avoided by performing this same computation on all processors.

As just indicated the cell driver supplies the main program on the nodes. The application level program on the nodes consists of five functions which are called by

the cell driver node program. These are:

- *Compute_Map*
- *Initial_Setup*
- *transition_function*
- *Convert_To_Display_Image*
- *setup_for_next_iteration.*

We will now look at each of these functions in turn and see how they fit into the node component of a cell driver application.

4.2.3 Compute_Map

Compute_Map is the function that determines the partition of the input raster among the processors. It is actually part of the host program as well as the node program, since it can be run either on the host or the nodes. If the *Compute_Map* is run on the host, the driver will export the global map data to the node programs, but it is expected that normally it would be faster to run *Compute_Map* independently on all of the nodes. The output of the *Compute_Map* function consists of the coordinates of the lower left corner of the raster component associated with each processor together with the height and width (in cells) of the raster component. This fully characterizes the partition.

If the application can run with a raster that is partitioned roughly evenly among the processors, the writing of the *Compute_Map* function can be avoided. The *Default_Compute_Map* function which is part of the driver will compute the evenly balanced partition, taking into account the block structure which may be built into the raster. In this case, the user must supply a *Compute_Map* function that does nothing but report an error if it is called.

If an unevenly divided raster is required, the developer of the application must write the *Compute_Map* function. In this case the *Default_Compute_Map* function is

Figure 4.1: Behaviour of the *Default_Compute_Map* Function

provided as a useful example for developers. Any *Compute_Map* function must set up the *Global_map* global variable and a number of other constants and flags.

For many applications it is expected that the behaviour supplied by the *Default_Compute_Map* (see Section 3.3.1) function will be suitable, in which case this chore can be avoided.

4.2.4 Initial_Setup

The *Initial_Setup* function is called by the application level of the node component of the cell driver after the global map is set up. Its purpose is to ensure that the cell driver data structures are set up correctly. This means that the initial iteration of the raster must be retrieved or created and the global pointer *Spatial_data* must be correctly associated with it. Cell driver functions will eventually be supplied to make this easier. At the moment, the default application provides an example of how it is done.

The *Initial_Setup* function must also set up any data structures that may be required to accommodate the exchange of data with neighbouring processors. The global pointer *Neighbour_data* is supplied by the driver as the expected location of neighbour data. This enables the other user written functions to gain access to the

neighbour data and also acts as the location used by the default neighbour handling functions that are included in the driver.

If the users are able to use the *Default_Compute_Map* function it is highly likely that they will also be able to use the *Default_Setup_Neighbour_Info* function as well as the other default neighbour data handling functions that are provided as part of the cell driver.

The *Default_Setup_Neighbour_Info* function assumes that the neighbourhood is symmetrical about the affected cell and that at most eight nearest neighbour processors are involved in the exchange of neighbour data. This should include a very large subset of two dimensional CA applications as discussed in Section 3.3.2.

Unless the global variable *Neighbourhood_flag* is set to VONNEUMANN, the default neighbour handling functions will exchange data with the eight nearest neighbours. The exact amount of data exchanged will vary according to the size of the neighbourhood around the central cell and the number of bytes in a raster cell, as well as the height and width of a the raster block that is assigned to the local processor.

Having established the required data structures and initialized the raster, the *Initial_Setup* function must then exchange data with the neighbouring processors that are affected by the state of the local raster component and those that affect the state of the local raster component. For a symmetrical neighbourhood, the processors affected and the processors affecting will be the same ones.

With a symmetrical neighbourhood centred on the affected cell, there are functions built into the driver to handle the exchange of neighbour data. These functions are *Default_Send_Neighbour_Data* and *Default_Get_Neighbour_Data*. They should be used in conjunction with the *Default_Setup_Neighbour_Info* function which we have just discussed.

The *Initial_Setup* function also should set the initial value of the *Display_this_iteration* flag and the *Output_this_iteration* flag. These flags tell the driver respectively whether to display the current iteration in a window and whether to write the current iteration out to the raster manager.

The *Initial_Setup* function will also set the value of *Neighbourhood_flag*. Currently the possible values of *Neighbourhood_flag* are VONNEUMANN and MOORE.

Eventually this flag will also have `USER_DEFINED` as a possible value, and the neighbourhood will be specified in another variable as a byte string.

The *Initial_Setup* function can also set up values for *Temporal_data* and *Attribute_data* if these are required by a given application. The user is also free to do any other initialization that is needed by the specific application.

4.2.5 The transition_function

The transition function is at the heart of a cellular automaton application. This function determines how the CA evolves from one generation to the next. It must reference the *Spatial_data* and the *Neighbour_data*. Eventually there will be default functions to reference the nearest neighbours of an arbitrary cell. For now, the transition function in the test application provides a good example of how to reference the spatial data and neighbour data correctly.

The transition function is expected to compute the next state for each cell in *Spatial_Data* and write the result to the corresponding cell in *Output_data*. At the end of an iteration the driver will swap *Spatial_Data* and *Output_Data*. So if previous iterations need to be kept in memory, they will have to be copied to *Temporal_Data* in the *setup_for_next_iteration* function (see Section 4.2.7). Functionality to manage temporal data will be added to the driver eventually. The user will specify the number of generations to keep with a flag and the driver will take care of it.

Since cellular automata applications typically use simple localized behaviour to model complex emergent behaviour, the *transition_function* may well be very simple to code, but this will depend on the particular application.

The driver has a number of global variables that are expected to be of use to the transition function. These include

1. *block_height* the height in cells of the raster block on the local node
2. *block_width* the width in cells of the raster block on the local node
3. *CellSize* the size in bytes of a raster cell
4. *row_size* and *col_size* the number of processors in a row/column of the mesh

5. *myrow* and *mycol* — the row/column that the local processor occupies in the mesh of processors
6. *mynode* — the NEMO logical node ID of the local processor
7. *map_ptr* — points to the global map entry of the local node

4.2.6 Convert_to_Display_Image

Every cellular automaton has its own particular raster data and the raster data base manager provides a facility for creating and retrieving rasters with data structures which are convenient for each application. The display manager has its own way of representing the graphical images that appear in its windows. Normally if a user program wishes to display some results it will have to map the data to the display manager's colour map before the data is sent to the display manager. This is the intended purpose of the *Convert_to_Display_Image* function. This function is written by the user and will be called by the driver before the display data is sent to the display manager. Thus it will be called on any iteration for which the *Display_this iteration* flag is set to true.

When the driver sends data to the display manager it expects to find the data at the global pointer *Display_raster*. For now it is up to the user to allocate a sufficient amount of space the first time this pointer is used. The required amount of space is presently one byte per cell though the display manager contains provision in its interface (but not yet in its functionality) for other possibilities.

Each cell of the raster in *Spatial_data* should be mapped to a colour value in the corresponding byte of the *Display_raster* structure. The *Convert_to_Display_Image* function in the test application provides a good example of what is required.

In addition to the data conversion the *Convert_to_Display_Image* function sets up some header data to tell the display manager where in the display window to display the data from this processor. This information is based on the position of the local raster component in the global map.

4.2.7 The `setup_for_next_iteration` function

This function is actually a subset of the *initial_setup* function. It does all of the initialization that is required for every iteration, such as the exchange of neighbour data, and the setting of the raster display and raster output flags, but not the allocation of data space that needs to be done just once.

If the exchange of neighbour data in *Initial_Setup* was done with the default functions, this can also be done here. If a custom neighbour data exchange procedure must be developed for the *initial_setup* procedure, the same procedure should also be reusable here.

The calling of `setup_for_next_iteration` is the last action performed by the cell driver in every iteration. It is intended to ensure that the data structures are all set up for the next call to the transition function.

4.3 Integration with other NEMO Components

As part of NEMO, the cell driver benefits from access to two utilities that were implemented to provide common services. These are the Display Manager (DM) and the Raster Database Manager (RPM). This meant that the cell driver did not have to manage the communication of data between the host program where images are displayed and raster files are maintained.

The DM and RM both offer an API to programs running on the AVX nodes that allow these programs to open and write to files and to display data in a window.

In the course of integration with the Display Manager, one issue that arose was the need to synchronize display from the various processors. Since each processor in the demonstration application (see Section 4.4) manages a separate piece of the raster and displays that piece independently, there is a need to coordinate these displays so that an iteration is not displayed until all processors have displayed and to ensure that all the patches in the display come from the same iteration.

In order to make this possible, a double buffering system was added to the display manager. The processors are able to write their images to a buffer that is not

displayed. The host processor is notified by each of the node processes when their display is done. When the host process has heard from all the processors it makes a call to the display manager to move the hidden buffer to the foreground.

The display manager added a call that would allow the node programs to do a blocking display call. This call does not return until the display manager on the host has finished writing the image from that node to the buffer. After they notify the host that they have displayed, the node programs are free to continue with the computation of the next iteration but before they display again they look for an acknowledgment from the host that the previous iteration has been swapped out of the hidden buffer.

Access to the raster manager makes it possible to save the results of a cell driver computation as a raster file. The cell driver allows the developer to save any or all iterations by setting a flag. The *create_raster* function which must be written by the developer permits the creation of a raster file before the computation of the cellular automaton begins.

In addition to the RM and DM, NEMO also provides a user interface process. The user interface allows the users of NEMO applications to specify the size of the mesh on which an application is to be run and to set the values of the command line parameters to these applications. The user interface provides a standard set of command line parameters to all cell driver applications. In addition it provides calls that allow the applications to prompt for user input using a dialogue box.

Access to these services reduced the effort needed for driver development and allowed the developers of the drivers to focus on the particular mission of each driver.

4.4 Demonstration Application

The Cell driver is intended to facilitate the porting of cellular automata applications from the sequential to the parallel worlds. An important part of the development process was the design and implementation of a test application. The test application is based on a paper by Per Brinch Hansen [25]. In this paper, a simple fire propagation model is described based on the principles of cellular automata.

The Brinch Hansen fire model is not an attempt to undertake the complex task of modeling fires under realistic conditions. It is a highly idealized process, not unlike the Game of Life described in [22]. In the simple fire model of Brinch Hansen, cells can have one of three states. They are either alive, burning or dead. Live cells spontaneously become burning cells on each turn with low probability signifying fire starting events such as lightning strikes.

Once a live cell has a burning neighbour, it becomes burning at the next generation. Burning cells unconditionally become dead cells after one generation of burning. Finally, more to keep the model going than to do anything realistic, the dead cells become live cells with a certain very low probability. The net effect is not very life-like, but serves to clearly illustrate the principles of cellular automata which is Brinch Hansen's purpose.

We implemented the Brinch Hansen model to provide a sample cell driver application. Since the Brinch Hansen algorithm uses the Von Neumann neighbourhood, exchange with only four neighbours was required. Therefore the four neighbour data exchange case was implemented first, and then extended to eight neighbours to produce the default neighbour passing behaviour that we have discussed in the previous section.

We modified the Brinch Hansen algorithm by introducing several fire states and having a burning cell progress from one state to the next with 50% probability at each iteration. We also modified the fire spreading behaviour so that a cell with a burning neighbour has a strong probability rather than a certainty of igniting when a neighbour is in one of the first three burning states (there are five in all). We displayed the output using the NEMO display manager. Some selected iterations are shown in Figure 4.2.

Besides using it as a test bed, we were interested to use the test application to confirm that the addition of parallel computing power to the Brinch Hansen algorithm would really lead to better performance as the number of processors increased.

The implementation of the demonstration application has served as a proof of concept of the cell driver. It demonstrated that the cell driver facilities can correctly coordinate the computation of a cellular automaton application distributed over a

Figure 4.2: The Display Output of the Demonstration Application

number of processors. Indeed the demonstration application will run on any rectangular mesh. In Section 6.4 we see the timing results of running the demonstration application on square meshes containing from 1 to 64 processors. The results also show the response of the application to different problem sizes.

The demonstration application also serves as a model for realistic applications. Currently, an earthquake modeling application is under development by Zhengping Wang.

Chapter 5

XAS: Goals, Design and Development

One of the system tools that seemed to be needed for effective parallel development on the Alex AVX II was a simulator that would run on the Sun workstations that we use as our sequential platform. This would enable us to debug using the familiar tools of the Unix and X-Windows environment as well as to easily trace all of the message passing behaviour of a simulated AVX application.

The simulator we developed is called the eXtended AVX Simulator (XAS). Its design and development was described in detail in [44]. Here we present a slightly condensed description of this work.

5.1 Motivation for the Project

In this chapter we describe the development of a simulator for the Alex AVX II parallel computer. The extended AVX Simulator (XAS) is the successor of a program called the Simple AVX Simulator (SAS). SAS was developed by J. Neilson of the School of Computer Science at Carleton University and slightly extended by the author. SAS emulates the point to point network level communication of the Alex AVX II computer.

In addition, SAS emulates a number of network information commands. These

commands allow programs to know their location in the simulated network as well as the size of that network and the address of the node from which the network was booted. SAS is useful as a prototyping tool for programs that communicate using only the network messaging functions of Trollius. It also has value as a pedagogical tool in introducing beginner Trollius programmers to a simple subset of the Trollius system calls. SAS also provides a tracing mechanism that allows the message passing activity of a parallel program to be monitored.

The hardware emulation in SAS is very simple. It models the dual processor compute nodes of the AVX as two Parasol nodes (some background to Parasol is given in Section 5.2.3). However, the coupling at the process scheduler level between the i860 and T805 nodes is not modeled. SAS does not contain any representation of the network topology and thus does not model the process of message routing. In addition, it lacks any representation of the lower level messaging functions of Trollius and has no support for the multi-casting and multi-reeling capabilities of the network layer.

XAS is a second generation simulator that provides a more realistic model of the behavior of the AVX. The XAS project has the following goals:

1. Provide a development platform other than the AVX itself that supports all of the commonly used system calls of Trollius. This will allow developers to produce complex applications on the simulator that may then be ported to the AVX at a later time.
2. Provide a better debugging environment than the machine itself. Lack of good debugging facilities is a common deficiency in parallel computing.
3. Provide a system that will allow classification of program errors as AVX hardware, or software problems, or program level bugs.
4. Provide a correct model of the message routing that occurs in the real machine, so that important factors such as network congestion and message buffering can be modeled in the simulator.

5. Add support for Trollius multi-casting and multi-reeling so that programs using these features can be developed and debugged using the simulator.
6. Address the issue of coupling on compute nodes between processes on the i860 and the scheduler on the T805. In the real AVX this coupling is handled by a software agent process running on the T805. This will ensure that the ordering of local messages between the i860 on a given node and the local transputer and beyond is performed correctly.
7. Provide a legacy system in which software written for the AVX will always run in a unix environment. This will ensure that such software will always be runnable, even if the AVX running Trollius is superseded by an incompatible successor.

The remainder of this chapter describes the design and development of the extended AVX simulator (XAS). In Section 5.2, we provide some background on the systems used to develop the simulator. These systems include the Alex AVX series II parallel computer, the Parasol modeling environment, and SAS, the simple AVX simulator which was the predecessor of the simulator described in this paper.

In Section 5.3 we provide an overview of the XAS system. In Section 2.1.2 the relevant components of the Trollius operating system are described and their adaptation to the XAS environment is detailed. Finally the concluding section contains a discussion of unresolved issues and suggestions for further work.

5.2 Background to the XAS Project

The XAS simulator was built on top of the Parasol 2.2 distributed modeling environment. Parasol supplies the emulation of the hardware, an interprocess message passing primitive and process launching capabilities. The general approach of the development of XAS is to reproduce the behaviour of the operating software of the AVX on the emulated hardware.

5.2.1 Simulation of Parallel Computers

Considerable work has been done on simulation of parallel computers, with a number of different aims. One very important goal is to facilitate the design of new hardware platforms by modeling the behaviour of the hardware. For example, software has been developed to study the behaviour of different interconnection networks [21]. A system for modeling processor behaviour is described in [42]. Other studies aim to speed up the performance of simulators of parallel machines by computing execution times from examination of the code ([20]).

The XAS project concentrated on accurately reproducing the behaviour of the operating software of the Alex AVX II system, using a hardware model that approximates the actual AVX hardware. The purpose of the project is described in Section 5.1. Performance prediction of the type that is described in [21], [42] and [20], would be an interesting extension of XAS. Some work has begun in this direction.

5.2.2 The Alex AVX II Parallel Computer

In this section we expand on the overview of the Alex AVX II given in Section 2.1.2. The Alex AVX series II runs under the Trollius Operating System. We had access to the source code of Version 2.2 of the Trollius operating system, and this has made the development of an accurate simulator possible.

The AVX II is a Multiple Instruction set Multiple Data stream (MIMD) parallel computer which can have up to 64 dual-processor nodes with a configurable network topology. While the AVX can be configured with several types of nodes, (such as special purpose graphics and mass storage nodes) the machines at Carleton University are mainly equipped with the typical compute nodes, which consist of a pair of processors:

- one Intel 860 (i860) processor (used for computation)
- one T805 transputer processor (typically used for communication and to run the Trollius Operating System)

Each compute node is equipped with up to 64 megabytes of random access memory that is shared by the two processors and up to an additional 16 megabytes that is reserved for the use of the transputer.

The Alex AVX II is a product of Alex Informatique of Montréal. It comes in a number of configurations including 16, 24, 32 and 64 node versions. The node counts refer to user nodes known as "inside the box nodes" (ITB nodes) that can be allocated by users for their exclusive use and linked together in a topology of the user's choice. Each node can be connected to up to four others by means of the user configurable network. A connection network commonly chosen by AVX users is the 2 dimensional mesh, but other topologies, such as a torus, pipeline and ring are also supported.

In addition to these nodes, each AVX II machine has a host node, which serves as a platform for running UNIX based configuration management tools as well as user programs. The host node is a SPARC processor running the Solaris 2.4 operating system. On the host Trollius consists of a collection of UNIX processes. Each machine also has a number of root nodes. A root node serves as a connection between the host node and the other ITB nodes allocated by the user. The presence of the Trollius operating system (see Section 2.1.2) on all nodes provides a common interface for user programs.

Other than the host, every AVX II node has a T805 transputer, which serves to run the Trollius operating system and to manage the communication into and out of each node. In addition to the transputer, AVX nodes other than the root node and host node, have another processor resident. As described earlier in this section, most of the nodes in the AVX II machines at Carleton are "compute nodes". The second processor on a compute node is the i860. The i860 shares up to 64 megabytes of memory with the transputer and provides a fast computing engine. The transputer runs the Trollius operating system.

While the transputer supports a multitasking scheduler, the i860 will only run a single process at any one time. When a process on the i860 sends a network message (see discussion of the Trollius operating system in Section 2.1.2.) the message is handed off to an agent process running on the T805. The agent process is scheduled by the transputer kernel. In this way processes running on the i860 come under the

control of the scheduler on the T805. See Section 5.4.8 for a discussion of how this behaviour is emulated in XAS.

The Trollius operating system runs on all AVX nodes, including the host. The emulation of the message passing features of Trollius is the purpose of XAS. These features are introduced in the next section.

5.2.3 The Parasol Modelling Environment

Parasol was designed for and is particularly well suited to the support of modeling computer systems. It specifically was aimed towards modern distributed and/or parallel computer systems in which the software as well as the hardware involved must be adequately represented¹.

Parasol is used in XAS to emulate AVX processors as well as to provide a message passing primitive upon which the emulation of Trollius messaging can be built. It is a useful foundation for a parallel simulator because it contains already realized mechanisms for

- the simulation of processing nodes with a choice of preemptive and non-preemptive process scheduling policies, with or without round-robin
- the launching of tasks (processes) on these nodes based on user written functions
- passing arbitrarily long messages between tasks

The availability of a choice of process scheduling policies is a useful feature for AVX simulators, since the AVX contains single process i860 processors, multitasking T805 processors and multitasking round-robin SPARC processors.

5.2.4 SAS Simulator

In SAS, the AVX hardware is modeled using Parasol nodes. The user determines the number of compute nodes as a command line parameter to a SAS application. In addition to the compute nodes, the system automatically creates a root and a

¹Taken from [37]

host node. All nodes can be used to run user tasks (processes). The simulated compute nodes each have two processors, an A860 and an A805 (see section 2.1.2 for an introduction to AVX compute nodes). The user can launch tasks on either the A805 or A860 processors associated with each compute node. The root node has just an A860 processor and the host node has a SPARC. User tasks can be run on these processors as well.

At startup time, SAS launches a *communicator* task on the A860 processor on each node. This task acts as a simple message matching kernel for SAS. The communicator task on a given node receives all network receive (*nrecv*) calls issued by user tasks running on the local node and receives all network send (*nsend*) calls issued by user tasks anywhere on the simulated network and directed to the node they are running on. Thus the system delivers *nsend* calls directly to the destination node. The communicator task implements the Trollius network message passing policy for matching *nsend* and *nrecv* requests (This policy is described in section 5.4). Send and receive calls are buffered in the communicator task and when a send and receive call are matched, the message of the sender is delivered to the receiver. Thus, a very simple emulation of the Trollius network message passing semantics is achieved.

There is a global data structure that keeps track of the variables that are needed by SAS on each simulated node. These variables include the Parasol port and process ID of the local communicator task and the Parasol processor ID's of the processor or processors associated with each SAS node.

As in Trollius, there are both blocking and non-blocking receive calls as well as a blocking and non-blocking send. In Trollius, the blocking send will only block if the system is unable to take on the sender's message for forwarding. Since this rarely happens in practice, the blocking and non-blocking sends almost always behave the same way in Trollius. In SAS the blocking send blocks until the message is delivered and the non-blocking send returns without delivering the message if a corresponding *nrecv* has not already been received at the destination node. This constitutes a difference in behaviour from Trollius. (In SAS version 1.1, to be released soon, the behaviour of the *nsend* function is the same as in Trollius.) In Trollius as well as in SAS, the blocking and non-blocking receives represent different behaviours. The

blocking receive will block the caller until a matching message is delivered. If a message is waiting to be delivered the receive call returns immediately with the message buffer that was sent. Otherwise the receiver remains blocked, possibly indefinitely.

If there is a message waiting to be delivered, the non-blocking receive will return *OK* and put a message in the receiver's message buffer. If there is no message waiting the nonblocking receive will return with an error code indicating that a regular receive would have blocked. In the first case the behaviour is the same as that for the blocking receive, but in the latter case, the receiver continues to run rather than blocking.

Users of SAS link their own task code (written in C) with the appropriate SAS and Parasol libraries. In addition they must write an *al_setup()* function. *Al_setup()* uses the SAS function *al_launch* to launch user tasks on the processors of the simulated AVX nodes. SAS itself creates the simulated network, launches the communicator task on each node, and then runs *al_setup*. Parasol handles the scheduling of tasks on each of the simulated nodes and the multitasking and timing of the various processors according to a single global clock. Details on the use of SAS are available in [45].

Tasks in SAS synchronize with the global clock by calling *al_tic()*. *Al_tic()* causes the time associated with the task to be advanced by 1. This may cause the task to be swapped out by the Parasol kernel until all other runnable tasks advance their clocks to the same point. In SAS, timing is very simple. The time associated with a task is advanced whenever a message is delivered. Thus it takes the same time to deliver a message from any node to any other. In effect, the SAS nodes are all part of a fully connected network over which messages of any length are transmitted in the same amount of time. While not realistic, this affords the developers an environment in which the logic of the local tasks can be developed and debugged and network message passing used without worrying about synchronization and timing issues. Once the logic of local tasks is working the program can be moved to the real machine for further testing and refinement. Alternatively, the program can now be moved to XAS.

The principle features of XAS are discussed in Section 5.3.

5.3 Overview of the XAS Program

5.3.1 XAS Data Structures

Like SAS, XAS is written in C. It is founded on a series of global data structures that represent the simulated XAS nodes and links. These data structures are summarized here and presented in their entirety in the appendices to [44].

For each AVX node modeled in a given simulation, there is an *xnode* structure which is used to cache the ports and process ID's of the various tasks that make up the emulated Trollius operating system. In addition, each node has four *link endpoint* data structures. These are used to cache the ID's of the neighbouring nodes associated with each link. In addition, the Parasol ports of the Trollius tasks that receive communication over the links are cached in these *link endpoint* data structures so that the tasks that send data over the links can find them.

In addition to the *xnode* data structures just mentioned, there are other per-node data structures associated with the buffer daemon, router daemon and cast daemon on each node. The buffer daemon, router daemon and cast daemon are discussed in sections 5.4.3, and 5.4.5. There is also a global virtual circuit table. (Virtual circuits are discussed in Section 5.4.1)

5.3.2 Basic XAS Functions

XAS offers users the ability to launch tasks on any of the simulated nodes. This is accomplished using the *xas_launch* function. The user must specify a processor name when launching tasks on a given node, since some nodes have more than one processor. (In the real machine the loader can determine from the type of executable which processor it should be run on.) The names we have given to these processors are reminiscent of those in the AVX, but at the same time not exactly the same to remind the user that the hardware being used is the XAS simulated hardware. The simulated compute nodes each have an *X805* and an *X860* processor. To be consistent the user must also specify a processor name on the single-processor root and host nodes. On the root, the name specified must be *X805*. On the host the

name given must be *XPARC*.

In addition to *ras_launch*, there is an *ras_tic* function that works similarly to the way the *at_tic* function works in SAS. Tasks that call *ras_tic* advance the time associated with the task, consuming processor time as they do so. If there are other tasks whose time is not so far advanced, the caller of *ras_tic* will be swapped out by the Parasol scheduler and some other task will be run.

5.3.3 XAS Tasks

When an XAS application starts up, the main task creates the necessary data structures, starts up a number of XAS tasks on each node and finally runs the user-written *ras_setup* function which launches user tasks on the processors of the XAS nodes.

Like Trollius itself, XAS is built up out of a number of intercommunicating daemons or tasks. Each daemon is responsible for its own component of the data communication process. These daemons implement the various communication layers in Trollius and themselves use these layers.

On each XAS node, in addition to any tasks launched by the user, the XAS system will launch a kernel communicator (to support kernel layer messaging), four data link input daemons and four data link output daemons, (to manage communication over the simulated links), a buffer daemon, router daemon and cast daemon (to implement data link and network level messaging), and an i8agent process (to synchronize tasks running on the X860 node with the scheduler on the X805, where the Trollius system tasks are running).

In Section 5.4, we give a detailed description of the Trollius communication layers and the role that each of the Trollius daemons plays in implementing these communication layers. At the same time we show how each layer is emulated in XAS and discuss the emulation of the corresponding Trollius daemons as XAS tasks.

5.4 Trollius Operating System Components and their Emulation in XAS

In order to meet the goals described in Section 5.1, XAS must model the behaviour of the Alex AVX II running Trollius in much greater detail than SAS does. In particular, the full details of the message passing interface offered by Trollius are implemented in XAS. This will allow developers to produce programs on the simulator using the message passing layer that is most appropriate for their application. The presence of all layers also allows us to incorporate tracing into XAS that monitors all phases of the message passing process. This tracing can be turned on and off by the user for any given run of an XAS application. It can also be targeted to the particular Trollius communication layer that the user wishes to monitor. (Trollius communication layers are discussed in the remainder of this section.)

There is an overview of Trollius in Section 2.1.2. Here we explain the components of Trollius in more detail and discuss how each is emulated in XAS.

5.4.1 Emulation of Trollius Kernel Level Messaging in XAS

The kernel layer in Trollius offers local message passing services to both system and user processes. There is a blocking send (*kscnd*) and a nonblocking send (*ktry_snd*). Similarly there is a blocking receive (*krcv*) and a nonblocking receive (*ktry_rcv*). The blocking calls do not return until the local Trollius kernel has completed the delivery of the message. The nonblocking calls will complete message delivery if there is a corresponding blocking call already waiting. Otherwise, they return without a message being exchanged. Communication at the kernel level is achieved by means of ports. Each Trollius process has a public port that is normally used for communication with the kernel.

This behaviour is very similar to that implemented by the SAS communicator process and the SAS communicator was the starting point for the kernel message matching task in XAS, which is called the kernel communicator.

Trollius message passing is implemented using an arbitrary integer called an event

Figure 5.1: Kernel Communication in Trollius and XAS

and a second integer called a type. Senders do not refer to a target process directly. Instead they specify an event. Receivers likewise do not specify a sender but rather an event. In order for the kernel to deliver a sender's message to a receiver, the events specified by sender and receiver must be the same.

Furthermore, both sender and receiver must specify a type. If either the sender or the receiver specifies a type of 0, and if they both specify the same event, then the message is transmitted. But if the sender and receiver both specify a non-zero type then, in addition to the requirement that the events be identical, the bitwise AND of the sender's and the receiver's types must be non-zero for the message to be transmitted. Basic kernel level communication is depicted in Figure 5.1.

The kernel communicator task implements the Trollius kernel message passing policy described above in XAS. All calls to *ksend* and *krecv* result in a Parasol message being sent to the kernel communicator task. The kernel communicator keeps separate queues of send and receive requests and each time a new send or receive request is received, the kernel communicator checks if the Trollius message matching rules are met. If a send and receive request are found to match, the buffered request is dequeued. The send request is acknowledged and the receive request is completed by

delivering the sender's buffer.

If a request does not match, it is either enqueued, in the case of a blocking request (*kscnd* or *krcvr*) or given a negative acknowledgment in the case of a nonblocking request (*ktry_scnd* or *ktry_rcvr*). The full behaviour of the kernel communicator process is described in the XAS system design document.

In addition to the behaviour just described, it is possible that a sender and a receiver wish to ensure that a series of messages are passed between two processes without the possibility that the receiver receives messages from another sender interspersed with those of the first sender. To ensure this, the Trollius kernel provides a mechanism known as a virtual circuit.

The virtual circuit is implemented by means of a private port that is maintained by the kernel. When sender and receiver wish to ensure the atomic transmission of a series of messages they both set a flag in the kernel message header. The kernel then establishes a private port that is used only for transmission using the particular event/type pattern of the initial message. Subsequent messages using the same event and type are then transmitted using the private ports. This ensures that packets from a different sender with the same event/type combination are not interspersed at the receiver with those of the intended sender. As long as sender and receiver continue to set the appropriate flag, the messages are transmitted on the virtual circuit. See Figure 5.2.

In XAS, the kernel communicator is based on the communicator task in SAS. The communicator task code was modified to work with the kernel message header (see Appendix A to [44] for a complete description of the XAS data structures) rather than the network message header. Then it was necessary to add the machinery for handling virtual circuits. Virtual circuits are emulated using a data structure similar to that of the actual Trollius kernel (The virtual circuit data structure used in XAS can be found in Appendix A to [44]). Like the Trollius kernel, Parasol supports a port based communication model. Since the establishment of virtual circuits is based on the allocation and use of private ports this mapped easily between the two systems.

In designing the virtual circuit data structure it was found that the complexity of the programming increases somewhat due to the fact that in the real AVX, each

Figure 5.2: Virtual Circuits in Trollius and XAS

node has its own global data space, while in the Parasol environment, all tasks share a single global data space.

This means that whereas kernel tasks in the real machine maintain identically named global structures, we have to modify these structures in XAS in order to distinguish the data of one node from that of another. This issue cropped up again and again in the course of the design of XAS. It was necessary in each case to replace the global data structures with an array of similar structures that is dereferenced using the node ID of the XAS node that the caller is running on.

In the case of the virtual circuits, we replaced the kernel virtual circuit data structure with an array of virtual circuit data structures. We had access to the source code for Trollius *ksend* and *krecv* and where possible the XAS code followed that of Trollius. Modifications to the code were necessary to handle the modified virtual circuit data structures employed in XAS.

5.4.2 Emulation of Physical Messaging in XAS

Another major feature of XAS versus SAS is the modeling of the connections between neighbouring nodes. Connections are emulated by restricting the set of nodes that a given node can communicate with directly, to a subset of the complete node set. This subset constitutes the nearest neighbours. User topologies in the initial version of XAS are limited to 2-d meshes. With this restriction, the system setup uses the user-specified number of rows and columns to determine, for each node, which other nodes are its neighbours.

As in SAS, a global data structure is used to keep track of variables associated with each XAS node. (This structure is the *xnode* structure. See Section 5.3.1 for an introduction to this structure. The C description of the *xnode* structure is included in Appendix A to [44]). The *xnode* structure for each node contains four *data link endpoint* structures. Among other necessary items, the index of the neighbour node at the opposite end of the link is cached in these *link endpoint* structures at the time that the XAS network is created.

Once this structure has been established, inter-nodal messaging can be emulated using Parasol messages passed between the neighbour nodes. Access to the links in Trollius is normally restricted to the data link daemons, each devoted to sending or receiving messages over a particular link. These daemons are a key component in the implementation of the data link layer in Trollius and XAS. The caching of the neighbour node index in the data link endpoint structures at setup time enables the data link daemons to find their neighbours when they are created.

5.4.3 The Emulation of Data Link Messaging in XAS

As depicted in Figure 2.1 the data link messaging in Trollius is founded on the kernel messaging on the one hand and on physical messaging on the other. The data link layer functionality is furnished by data link input and output daemons, one of each for each link on a Trollius node. The data link input and output daemons begin life by registering with the Trollius kernel on the node where they are born and then input daemons begin to listen for input messages on their respective physical links

and output daemons begin to listen for kernel messages from local processes that want to send messages out over the links.

Communication between data link daemons and the other Trollius processes on their local node is done by means of the data link layer messaging functions, *dscnd* and *drcvr*. (There is also a *dtry_snd* and *dtry_rcvr* for nonblocking sends and receives). The *dscnd* and *drcvr* functions are built on top of the kernel messaging layer. They make use of a data structure known as the network message header. (See Appendix A of [44] for this structure.)

The sender's and receiver's of data link layer messages are identified by means of the *nh_dl_crent* field in the network message header. In the case of system processes such as the data link output daemons and the buffer daemon, there is a well known event associated with each that allows senders to reach them using *dscnd*. Each system process sits in a loop, repeatedly issuing *drcvr* requests using their well-known event as the value of *nh_dl_crent*.

A *dscnd* call begins by forwarding, with a *kscnd* call, the network message header associated with a data link message, setting up a virtual circuit as it does so. The network message header contains a pointer to a message body which contains the user's message buffer. This buffer is forwarded using a second call to *kscnd*. As this second *kscnd* call is done, the virtual circuit established with the first *kscnd* call is dissolved by resetting the flag in the kernel message header.

A *drcvr* call performs the other end of this transaction. That is, a call is made to *krcvr* to receive the network message header and establish a virtual circuit with the sender. A second call to *krcvr* is performed to receive the message body and take down the virtual circuit.

Since the *dscnd/drcvr* functions are implemented in Trollius using the kernel level messaging functions, and since the emulation of the kernel level messaging in XAS was completed and tested before the emulation of the data link layer, the Trollius code for these functions was ported to XAS almost without change. However, porting the data link input and output daemons themselves required more effort.

Each data link output daemon has its own well known data link event. It is this event which allows callers to *dscnd* to choose a particular data link output daemon

(and thus a particular data link) to send their message to. The data link output daemon uses *driver* with its own well known event to receive messages that local processes want to send off the node over its link. It then sends the message over the link to a corresponding data link input daemon at the other end.

The data link input daemon waits for messages to be sent over its link and then forwards them on using *nsend* (introduced in the discussion of the network layer in Section 5.4.5) either to another data link output process for further forwarding to a remote node or to a local receiver if the message is destined for the local node. This distinction is determined by the value of the *nh_node* field in the network message header.

However each daemon must send or receive over the physical link, and physical messaging is implemented differently in Trollius than XAS due to the difference between the transputer hardware interface on the one hand and the simulated parasol hardware interface on the other. Thus the daemons themselves had to be partly rewritten to run on the XAS platform. The Parasol messaging "hardware" was much simpler to use than the underlying transputer hardware of the real machine, but it was also necessary to add machinery to support the creation of virtual circuits over a link. This was done by adding a flag to the *link endpoint* data structure to notify the system if a virtual circuit exists over a particular link.

In addition to data link messaging, access to the Trollius buffering system is integrated with the data link layer. When a call is made to *dscnd* an attempt is first made to reach the appropriate data link output daemon, using *dtry send*. If this call fails a blocking send (*dscnd*) is issued instead to the buffer daemon for the local node. (As with the data link output daemons, the buffer daemon has its own well known event).

The buffer daemon will accept the message and hand off the delivery of the message to another process called a buffer worker. The buffer worker in turn makes a blocking send call (*dscnd*) and waits to be able to deliver the message to the intended receiver. This same mechanism also functions when a data link input daemon receives a message and attempts to deliver it to the local recipient (which can be either a local user process or another data link daemon depending on whether the message

is destined for a local process or to be forwarded on to another node). If the intended recipient is not waiting to receive a message at the time that the initial attempt to send is made, the message is instead passed to the buffer daemon which arranges for its later delivery using a buffer worker.

5.4.4 Data Link and Buffer Daemon emulation in XAS

As mentioned in the previous section 5.4.3, the porting of the data link daemons necessitated some design work because global data structures used by the data link daemons are different in the XAS emulation than in Trollius. The identities of neighbour nodes have to be cached when the simulated hardware is set up and then accessed by the data link daemons as they start running. Each data link daemon determines its link on a given node by reading a global variable. By calling *ras_lir* (see Section 5.3.2) in the system setup function, each time a new task is launched, the Parasol scheduler is forced to run the new task before the setup function continues its activities. Each new data link and buffer daemon reads a global variable to determine which node it is running on. In addition, the data link daemons read another global variable to determine which link is the one they are to control. Once the correct link is identified, the data link daemon checks the global data structures that are previously set up to learn the neighbour node that it will be communicating with.

Data link input daemons spawn a Parasol port and cache it in the appropriate link endpoint structure of the node that they will be receiving from. This allows data link output daemons to determine where to send their messages and represents the creation of an emulated XAS link.

The buffer daemons must set up some data structures that they use to cache buffered messages that are awaiting delivery. These data structures are global in Trollius but the shared Parasol memory forces us to modify them in the XAS environment. When writing the code to handle virtual circuits, the Trollius code was extensively modified to handle the new global data structures. In the buffer daemon, a different strategy was employed to deal with the modified data structures. Instead of modifying the Trollius code, C macros were declared to map the existing code to

the new data structures. While this resulted in more cryptic system code, it greatly reduced the effort involved in porting the Trollius buffer daemon code to XAS. Due to the success of this approach, the same strategy was employed in porting the Trollius router code when the network layer was added to XAS. This phase of the work is described next.

5.4.5 Emulating the Trollius Network Layer Code in XAS

The network layer send and receive functions work similarly to the data link layer in that they use the network message structure to specify the destination of a Trollius message. In addition to an event and type a user of a network send (*nsend* or *ntry_send*) must also specify a *destination node*. The *nsend* function passes this node to the router daemon to learn which data link must be used to forward the message to the remote node. (The router actually returns the well known event of the data link output daemon associated with the required data link.) The network send call then proceeds to use *dsend* to forward the message to the appropriate data link output daemon.

Another field in the network message header specifies the length of the user's message buffer. If this length is larger than some threshold (currently 4096 bytes) the *nsend* code will break the user's message up into two or more packets such that no packet is bigger than the threshold. The data link layer lacks this mechanism and thus data link messages are limited in size to less than 4096 bytes.

The *nrccv* call uses the message length field in the header to determine how many packets must be received to reconstitute the packetized message. As Trollius guarantees that messages from a given source to a given destination will arrive in the same order that they are sent, the packets are received in the correct order. Thus *nrccv* concatenates the packets in the order that they come in, to reassemble the message.

As it turned out ensuring that this ordering rule was adhered to required some effort in XAS. This issue is discussed in Section 5.4.8 which is concerned with the synchronization of i860 and T805 processes in the AVX and in XAS.

As with the data link layer, porting the code for the network layer send and receive

functions benefited greatly from the fact that the lower level communication functions were already emulated in XAS. As mentioned above, the network layer adds routing and large message packetization to Trollius. Message packetization ported without any modifications being necessary. Routing on the other hand, involved a major effort and the solution of some interesting problems.

5.4.6 Emulation of Trollius Routing in XAS

Routing in XAS was initially implemented using a simplified strategy. This strategy was to forward a message toward the correct mesh column and, once in the correct column, to forward the message toward the correct row. The rest of the *nsend* and *nrecv* code ported with few changes, as it is implemented in terms of *dscnd* and *drecv*. The substantial effort of this phase of the work involved the addition of the routing scheme as it is implemented in Trollius.

The router daemon in Trollius is started up when the system is booted and at first has no knowledge of the network. If asked the route to some node, it will always return the same data link event. However it has a call in its interface that allows the loading of a routing table. These routing tables are generated on the host and loaded into the ITB nodes by the Trollius network configuration tools when the user sets up a topology.

In order to emulate Trollius routing exactly, XAS runs the same network tools as are run in Trollius to generate the routing tables. The *trout* tool is then used as in the AVX to load the routing tables into the router daemon on each simulated node.

The router daemon offers a routing service that is used by *nsend* but is also available directly to all user processes. The router daemon in XAS begins by determining on which node it is to run, in the same manner as the data link and buffer daemons. The first call it receives will be an *rloadroute* call from the *trout* function. This call will load the routing table into the router daemon.

There is a chicken and egg problem in this loading process since the *trout* call itself uses the *nsend* function, while *nsend* determines the route of the message by consulting the local router. However, this problem is overcome by loading the routing tables

starting on the host node and using the already loaded routing tables to correctly load other routing tables onto nodes deeper inside the box.

The generation of routing tables on the host makes use of the Trollius *map* function. The *map* function in turn requires that a network configuration file known as a *bNail file* be generated. The *bNail file* contains a representation of the AVX topology, which must match the actual topology allocated by the user. For the generation of mesh topologies a script exists known as *mkmesh*. This script was developed by Peter Choynowski, the network administrator for the School of Computer Science. *Mkmesh* generates the required *bNail* file when the user specifies the number of rows and columns that are to be in the mesh. This code has been incorporated into XAS and is used to provide the correct *bNail* file so that *map* can generate the correct routing tables.

In this first version of XAS *map* and the modified *mkmesh* are run as separate UNIX processes during the set up of the simulated AVX hardware. Later versions should incorporate this functionality into the main XAS setup function to enhance the portability of the simulator.

In addition to the functions already described, the router daemon also allows the addition of new entries into an existing table. This function is used by the cast system when users create groups of nodes and define new addresses for them.

5.4.7 The Emulation of the Trollius Cast System in XAS

The Trollius cast system allows the sending of a single message to a group of nodes (a *multi-cast*) or the receipt of a single message from a group of nodes (a *multi-recv*). To do this a single node, (the *root* of the cast group) calls *rcast* and specifies an integer to be the cast ID and the nodes to be included in the cast group. Once the *rcast* call has completed, the cast ID can be used in the *nh node* field of the network message header of an *nsend* call to send a message to the entire group. The nodes of the group must all call *nrecv* in the usual way to receive the message.

Conversely, all of the nodes in the group can call *nsend* using the cast ID. The receiver calls *nrecv* and receives a single message which is the result of combining the

the senders' messages. The manner of combination is specified at the time of the *reast* call. Built-in combining functions in Trollius allow the *reast* call to specify that the messages are to be byte-wise added, AND'ed, or OR'ed, among others.

The implementation of multi-casting and multi-reeling is accomplished by means of the Trollius cast daemon. At the time that an *reast* call is made, a message is sent to the cast daemon on the root node. The cast daemon examines the list of nodes in the cast group and determines the links that lie on a shortest path to each of the nodes in the group. It then forwards a cast creation call to the cast daemons on the nodes at the other end of those links, passing the portion of the node list that lies in the direction of that link.

The cast daemons on the neighbouring nodes repeat the same process. In this way the cast creation calls are passed out from the root node in a tree-like fashion with the most distant nodes in the cast group as the leaves of the tree. As the cast creation message is passed to each node, the cast daemon on that node registers the cast ID with the router daemon on that node. The router daemon associates the cast daemon's well known event with the cast ID. The cast daemon caches the links associated with its portion of the cast group as well as the source link from which the *reast* call came.

When an *nsend* call is made from the root node, using the cast ID the event returned from the router daemon to *nsend* is the well known event of the cast daemon. *Asend* thus forwards the message to the local cast daemon. The cast daemon then passes the message over the links it has cached, to one or more neighbouring nodes where again the message is diverted to the local cast daemon. This process continues until all nodes in the cast group have received a copy of the message. Of course the messages will be delivered only if a process on each node in the group issues an *nrecv* call. The usual event and type matching rules apply.

In order to perform a multi-reel on the cast group, every member of the cast group must issue an *nsend* call using the cast ID. The senders set a flag in the network message header to specify that this is a multi-reel. The cast daemon will forward the message toward the root node of the cast group and, where more than one branch of the cast tree passes into the node the cast daemon will wait to receive from all the

branches. Once the cast daemon has received a message from all branches of the cast tree, it combines the messages using the combining function specified by the caller when the cast group was created, and forwards the resulting single message toward the root of the tree.

This process results in a single combined message being sent to the root node of the cast group. A process on the root node can receive this message by calling *nrccv* in the usual way.

Porting the cast daemon to XAS required that the data structures maintained on each node by the cast daemon be modified to take account of the fact that all cast daemons now existed in the same global address space. As with the buffer and router daemons, C macros were used to change references to the Trollius data structures to instead refer to the XAS data structures. Otherwise the cast code was quite self-contained and the effort involved in porting it was relatively small.

The one exception to this statement arose because of a bug that existed in the cast system code. This bug has to do with the fact that setting up a cast group is a distributed process that takes some time. If developers are not careful a race condition can exist in which processes may use the cast group before it is fully defined. The router daemon supplies a function that will block the caller until a cast group has been completely defined. This is the *nnedowait* function. A bug in this function was allowing callers to the multi-recv function to send their messages before the system was finished defining the cast group. As it turned out, the same bug existed in XAS when the cast code was first ported.

A bug fix was available for Trollius that ensured that the blocking worked correctly and the same bug fix worked in XAS.

5.4.8 Synchronizing Processes on the Simulated i860 and T805 of an XAS Compute Node

One of the more subtle issues in accurately modeling the Trollius system as it is implemented on the Alex AVX II arose in connection with the scheduling of messages transmitted from processes running on the i860 processor on a compute node of the

CHAPTER 6. SOME RESULTS OF THE WORK

Figure 6.5: Graph of Absolute Speedup vs. Number of Iterations on 64

what to expect in the way of absolute speedup, we must bear in mind that a single i860 node in the AVX can do just less than two iterations per minute. Sparcengine 20 that we used to run the sequential version can do 12 iterations per minute. Thus our parallel processors are running locally at $5/31.8$ or $16/102.4$ the speed of our single host processor in the computation of the values for an iteration. Thus if we get optimal speedup on 64 nodes we should see a speedup of 64 or 10.24. The actual values are shown in Figure 6.5. We can see that they are only 15% of optimal for large runs.

Given that the same algorithm is being run for the parallel and sequential versions (other than the way I/O is managed) it is legitimate to look at the relative speedup as well. To compute the relative speedup we extrapolated the values from Figure 6.3 using the formula $31.8x + 13$, to get an estimate of how long it would take the parallel program running on one node to perform the number of iterations that were tested with the parallel version running on 64 nodes. We computed the relative speedup using the estimated time of the parallel algorithm running on

AVX.

In order to ensure that messages passed to the buffer daemon are not overtaken by later messages received on a data link input daemon and forwarded directly to another data link output daemon, the buffer daemon and buffer workers run with a higher priority than the data link input and output processes. When sending processes are running on the simulated T805 (called an *X805* in XAS) this strategy is effective in preserving message order. However, if the sending processes are running on the simulated i860, (called the *X860* in XAS), the message ordering was not correct in spite of the higher priority given to the buffer processes over senders.

This was because the tasks running on the Parasol node associated with the *X860* were not subject to the scheduling priority of the Trollius system daemons running on the *X805*. As a result the sending process sent to the local kernel communicator before the buffer daemon, if the Parasol kernel happened to run the tasks on the *X860* node first.

Since this issue is a general one that should arise wherever two processors are running message passing processes that use the same kernel, it seemed certain that this issue had already arisen when Trollius was ported to the AVX itself. This turned out to be the case. In a private conversation, Greg Burns, the developer of Trollius, explained that a process called the *i8agent* was created to run on the transputer and that messages to be sent via the Trollius kernel are forwarded privately from the i860 process to the *i8agent*. The *i8agent*, which is subject to the transputer scheduler and which runs at a lower priority than the buffering code, then issues the call that transmits the i860 process' message to the kernel [39].

We adopted a similar strategy in XAS. An *i8agent* process runs on the simulated transputer. Before issuing a *ksend* call, the process on the *X860* exchanges an empty Parasol message with the *i8agent*. The *i8agent* runs with lower priority than the buffer processes. This brings the i860 process under the control of the transputer scheduler.

5.4.9 Summary of Trollius Processes Emulated in XAS

This completes the description of the components of Trollius that are emulated in XAS. In addition to the functions explained in the preceding sections, there is a setup function (as described in Section 5.3.3) whose role is to determine the size of the mesh that the XAS simulation needs to create, and set up the data structures for the simulated nodes, and for the XAS system tasks. The setup function then launches the data link daemons, the buffer daemon, the router daemon, the cast daemon and the *isagent* on each of the XAS nodes. Finally the setup function runs *ras_setup()* which the user must write and which launches the user's tasks to run on the nodes of the XAS simulated machine.

Figure 5.3 depicts the full set of system tasks that are created on each node by XAS.

With at least twelve XAS system tasks per node, and user tasks as well, the number of Parasol tasks created by a large (64 node) XAS application runs into the hundreds. This in itself was nothing new for Parasol. But as it turned out the task creation tree with hundreds of tasks directly beneath the root put unprecedented demands on Parasol and revealed a weakness in the Parasol 2.2 system in which our implementation began. Fortunately an upgrade of Parasol to version 3 was under way at the time of XAS' development. The bug was quickly found and rectified by Patrick Morin and the improvements resulting from this change incorporated in the latest version of Parasol.

Another Parasol feature identified as a result of XAS development was the need for separate global address spaces for each Parasol node. These *glocal* variables were also incorporated into Parasol version 3, which was officially released in the fall of 1995.

AVX NODE

Data Link Input and Output Demons

Figure 5.3: XAS System Tasks

5.5 Conclusion and suggestions for further work on XAS

We have implemented a simulator that contains the complete communication behaviour of the Alex AVX series II parallel computer running Trollius. In addition to a useful tool this work has provided us with deeper insight into the details of the system software of the Alex AVX machine that we are using for parallel application development.

Future enhancements to XAS should include the addition of a graphical user interface and the ability to handle the source code of Trollius applications without any modification. At present, it is necessary to pass in any command line parameters from the *xas_setup* task by means of the simulated messaging routines.

In addition to a graphical front end, a graphical back end to display the results of XAS tracing would also be very useful. The volume of information produced by XAS tracing can be large and graphical tools would be useful for extracting information from the trace data more easily.

In addition to developing applications, it would be interesting to use XAS to test modifications to the Trollius operating system itself. These modifications could be used to enrich the semantics of the message passing calls currently supplied by Trollius. This work would be useful in the implementation of the MPI standard at the system level.

The addition of realistic timing to XAS would make it useful as a performance predictor. Measurements of the message passing performance of the AVX are being made as part of the AVX timing project. It is hoped to incorporate these measurements in the next release of XAS.

Chapter 6

Some Results of the Work

6.1 A Note on Measuring Parallel Performance

In describing the benefits of parallel computing the key measurement is the *speedup*. Suppose we have a problem P that we want to solve in parallel. Suppose further that we have a sequential program for the solution of P . If we let T_s be the time for a sequential solution to problem P and time T_p be the time for the parallel solution on p processors to problem P then the speedup S_p on p processors is defined in Equation 6.1.

$$S_p = T_s/T_p \quad (6.1)$$

This may seem straightforward, but the question arises as to which sequential algorithm to use. Ideally, T_s should be the optimal sequential time. If no optimal algorithm is known, we may use the fastest available. But if there is a faster sequential algorithm than the one that is chosen, then the true speedup will be less than the computed speedup.

Even if we use an optimal sequential algorithm, the question arises as to which machine to run it on. Available sequential and parallel machines generally use different hardware for the computation. If one has state of the art parallel and sequential machines, the processor in the sequential machine will often be faster than the processors in the parallel machine.

For the sake of comparison, let us call the speedup defined in 6.1 *absolute speedup*. For T_s , we will use the time of the best known sequential algorithm running on the fastest platform to which the sequential program has been ported. This is the speedup that is of interest to the application developer. This person wants to know how much faster the problem can be solved using the parallel program on the available parallel machine than it can be solved on the available sequential machine.

Now let us consider an alternative definition, which we will call *relative speedup*. Let us suppose that we have a parallel program that will run on any number of processors, dividing the work among them.

Let us define relative speedup on p processors S_p , as in equation 6.2

$$S_p = T_1/T_p \quad (6.2)$$

In the case where there is slower hardware in the parallel machine, disappointing absolute speedups may conceal the fact that a parallel algorithm has been developed that *scales well*. By this, we mean that we can get increases in performance by adding additional processors. If an algorithm scales well we continue to get improvements in performance as we add more processors. (Note however that for a given problem size, there is always an upper limit to this process). Good scaling behaviour will be revealed if the relative speedup is close to n for all n in the range of processors available.

Relative speedups should be used with caution, since a poor parallel algorithm might scale very well but be so slow that a good sequential algorithm will solve the problem faster. On the other hand if poor absolute speedup is occurring mainly because of slower hardware, the relative speedup may well indicate that big improvements can be achieved when faster parallel hardware becomes available.

In this chapter, both absolute and relative speedups will be presented and we will be careful to distinguish one from the other. For a complete discussion of these issues, see [16].

6.2 CanBEIS

The parallel version of CanBEIS was developed in part to test the Alex AVX II communication. We introduced considerable additional output so that intermediate results of the computation could be displayed. On the other hand we replaced the reading of land-use data from files with hard-coded data. Thus comparisons between the sequential version and parallel version should be viewed with caution. The scaled down problem that we used in the development of parallel CanBEIS was run once on its original sequential platform and took approximately 900 seconds to solve the original problem for 31 days on 46 grid points.

After parallelization, this problem could be solved on 16 processors in 33 seconds for an absolute speedup of 27. Here we have a case where the difference in hardware produces a super-linear speedup. However if we ported this algorithm to the SPARC 20 which acts as the host in the Alex AVX II, we would be very likely to observe the opposite phenomenon. (See Section 6.3 for an example of this.)

For our small test problem on 46 grid points, the improvement in performance leveled off at 16 processors, at which point each processor was handling just two or three grid points.

Beyond this point, performance actually deteriorates as the cost of communication using many small messages overwhelms the benefit gained from further division of the computational work. These messages are a result of the additional output that was introduced in the parallel version of the program to allow us to display intermediate results.

To observe how our parallel implementation scales with larger problem size we tested again on 92 grid points and 184 grid points. We computed the relative speedups for each of the three cases at some selected numbers of processors from 1 to 64. Testing on twice as many grid points yields better relative speedups for 25 and 32 nodes after which the cost of communication again dominates and speedup declines with increasing numbers of nodes. When the problem size is doubled again, we observe that performance continues to improve up to 49 nodes. See Figure 6.1 for a graphical display of these results.

One interesting point is that we see slightly super-linear relative speedups for small numbers of processors on the larger problem sizes. This is probably due to inefficiencies introduced by packetization of the parallel communication. As a result of packetization, the cost of communication does not grow in a smoothly linear fashion but rather contains discontinuities at each point where a new packet is introduced. (This occurs with every increase of 4096 bytes).

Note that for the smallest case, the maximum number of processors engaged in the computation is 46, since the work cannot be more finely divided. At this stage, the communication cost dominates and we see no improvement between 32 and 64 processors. In the second case, we see that as the distribution of grid points approaches one per processor the speedup declines toward the same value as it does in the limit for the smaller case. Beyond 32 processors, the communication cost dominates and the performance declines. In the case where there are 184 grid points the observed speedup has begun to decline on 64 nodes at which point 56 processors are working on 3 grid points while 8 have two. If we had access to a machine with more than 64 nodes, we would probably observe a similar drop off in performance for the largest case.

If the processors are not in lock step and if the computation is not compute bound, there is also a saving in the actual time spent communicating, though this saving should be linear at best. When all results are computed on a single processor, they must be transmitted in sequence by that processor to the host. The time taken to send six or seven packets is greater than the time spent by four processors sending two packets each. Thus there is less time taken away from the computation and the overall time spent is less.

6.3 Clarke Fire Model

The sequential version of the Clarke Fire Model was ported to our system with minimal changes and run on a Sun SPARCengine 20 motherboard with superSparc 70 Mhz processor and 117 Megabytes of memory. We noted that this program used four 620 by 526 or 326,120 cell rasters. Each cell in these rasters is a short integer which

Figure 6.1: Speedups for Parallel CanBEIS

means that each occupies 652,240 bytes on our system. In addition, a fifth raster is used to accumulate the image results, for a total of 3,261,200 bytes dedicated to rasters. Of this space $4/5$ is global while $1/5$ is allocated from the heap.

Three of these rasters are loaded from files, one is computed from another file and the fifth is used for output. Of the four initial files, only one (the fuel raster) changes in the course of an iteration of the fire model. The sequential version that we received from Dr. Clarke, modifies the fuel raster in the course of an iteration. When a new iteration begins it reloads the original fuel values into this raster from the file. Since disk access is slow compared to memory access, we surmised that we could speed up the performance of the sequential version of the Clarke Fire Model by avoiding the access to disk at each iteration. We did this by storing the values for the initial fuel raster in memory and copying them into the working fuel raster at the beginning of each iteration. This copying step replaced the disk access in the original version.

For a run of 128 iterations the original sequential version took 959 seconds, while the modified sequential version ran 128 iterations in 658 seconds, an improvement of

with some problems means that there are good prospects for building many useful applications on the cell driver.

The proliferation of available parallel machines suggests that the benefits of techniques such as those described in this thesis should reach a growing population of users in the years ahead.

Figure 6.2: Performance of the Modified Sequential Version of the Clarke Fire Model

Figure 6.3: Performance of the Parallel Fire Model Running on one Node

Figure 6.4: Performance of the Parallel Fire Model Running on 64 Nodes

The differences must therefore be due to the fact that the parallel version is running on slower hardware. Under these circumstances, we look to the relative speedup to let us know how the parallel program would perform on faster hardware under similar I/O conditions.

The case we will consider for comparison is the performance of the parallel version running on 64 nodes for various numbers of iterations. Note that the overhead for loading data into and out of 64 nodes is not the same as it is for 1 node. There is some variation in the amount of overhead as well as differences in the amount of time that each node spends computing. As a result the scaling behaviour of the 64 node parallel version shows some scattering about the line that describes the time cost versus the number of iterations. (See Figure 6.4)

Given that the behaviour of the sequential version of the fire model as depicted in Figure 6.2 is linear, we extrapolate the performance we have measured using the formula $5x + 10$ and compute the values for the absolute speedup. The absolute speedup becomes close to optimal for larger numbers of iterations. When considering

Figure 6.5: Graph of Absolute Speedup vs. Number of Iterations on 64 Nodes

what to expect in the way of absolute speedup, we must bear in mind that while a single i860 node in the AVX can do just less than two iterations per minute, the Sun Sparcengine 20 that we used to run the sequential version can do 12 iterations per minute. Thus our parallel processors are running locally at $5/31.8$ or $16/100$ of the speed of our single host processor in the computation of the values for an iteration. Thus if we get optimal speedup on 64 nodes we should see a speedup of $64 * 0.16$ or 10.24. The actual values are shown in Figure 6.5. We can see that they are within 15% of optimal for large runs.

Given that the same algorithm is being run for the parallel and sequential versions (other than the way I/O is managed) it is legitimate to look at the relative speedup as well. To compute the relative speedup we extrapolated the values presented in Figure 6.3 using the formula $31.8x + 13$, to get an estimate of how long it would take the parallel program running on one node to perform the number of runs that were tested with the parallel version running on 64 nodes. We computed the relative speedup using the estimated time of the parallel algorithm running on one node

Figure 6.6: Graph of Relative Speedup Versus Number of Iterations on 64 Nodes

and the measured times (as presented in Figure 6.4) for the parallel version running on 64 nodes. The values computed for relative speedup are shown in Figure 6.6. By comparing the curves of Figure 6.5 and Figure 6.6 it can be seen that the two speedup curves have a very similar shape. On a parallel computer with faster hardware, the absolute speedups would be bigger.

To summarize, we have shown that we can achieve useful speedups on our parallel platform. In the case of the Clarke Model, many iterations are needed to produce a statistically reliable result. Running on our available parallel platform, we can produce solutions that would take 5 hours to compute in about half an hour. Given a real time fire emergency, this time saving could be of great importance.

6.4 Brinch Hansen Fire Model and Cell Driver

As discussed in Chapter 4 the cell driver was implemented with the expectation that good speedups could be achieved with cellular automata applications running on

raster data. One reason for this rests on the following argument:

For a square neighbourhood the data that must be exchanged between neighbouring processors will be a strip around the perimeter of the piece of raster associated with the processor. The thickness of this strip will depend on the exact size of the neighbourhood. For a 3 by 3 neighbourhood the strip is one cell wide. For a 5 by 5, the strip is two cells wide. For an n by n neighbourhood, the strip will be $n/2 - 1$ cells wide. The total length of the pieces to be exchanged with up to eight neighbours will be equal to the length of the perimeter of the piece of the raster associated with the processor. Thus the amount of data to be exchanged is proportional to the length of the perimeter. (See Figure 6.7 for the example of the 3 by 3 neighbourhood).

Thus the amount of data that has to be exchanged is proportional to the square root of the number of cells in the portion of the raster that is associated with each processor. The local processing for the computation of an iteration of the cellular automaton is proportional to the total number of cells. Therefore, for some size of problem the cost of local communication will become small relative to the cost of computation.

With this in mind we measured the time cost of running the fire model application on various sized rasters and computed the relative speedups that we obtained. In running this experiment no I/O was done. Therefore this measurement compares only the computation and communication cost of the problem running on different numbers of nodes with the cost of running on one node (relative speedup).

The computation was run on three sizes of raster and we saw that parallel performance improves as problem size grows. The three sizes were a 512 by 512 cell raster, a 1024 by 1024 cell raster and on a 2048 by 2048 cell raster.

The results are shown in Figure 6.8, which compares them with the line $y = x$ which represents optimal speedup.

The smallest of the three test rasters contains 262,144 cells, while the largest raster contains 4,194,304 cells. For the small raster a nearly optimal relative speedup is obtained when progressing from one node to four. After this the relative speedup falls off as the processors get smaller subtasks and have to exchange data more frequently. For the 1024 by 1024 raster, good relative speedups (about 80% of optimal) are

Figure 6.7: The Perimeter Cells to be Exchanged for a 3 by 3 Neighbourhood

Figure 6.8: Speedups of the Demo Application for Two Different Raster Sizes

obtained using up to 16 nodes, after which the gains in performance level off, at a higher level than for the smaller problem. The largest raster shows even better performance. Even on 64 processors the relative speedup obtained is about 2/3 of the optimal value.

6.5 Conclusions and Further Work

We have demonstrated that GIS applications can be ported to a parallel machine and that useful speedups can be obtained. The parallel aspects of such programs can, to some extent be encapsulated in their own level. This should improve the ease with which parallel Cellular Automata applications as well as other parallel neighbourhood modeling applications can be developed.

There is improvement that can still be made to the API of the cell driver. This should further enhance its utility and ease of use. The great variety of useful cellular automata applications that exist, and the large amount of computation associated

with some problems means that there are good prospects for building many useful applications on the cell driver.

The proliferation of available parallel machines suggests that the benefits of techniques such as those described in this thesis should reach a growing population of users in the years ahead.

Bibliography

- [1] B. Alexander, D. Engelhardt, and A. Wendelborn. An overview of the adl language project. *Proceedings Conference on High Performance Functional Computing, Denver, Colorado, April, 1995.*
- [2] Carlo Arcelli and Giuliana Ramella. Finding grey-skeletons by iterated pixel removal. *Image and Vision Computing*, 13(3):159–167, April 1995.
- [3] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, June 1989.
- [4] M. Batty and Y. Xie. from cells to cities. *Environment and Planning B Planning & Design*, 21(S), 1994.
- [5] James R. Beers, Ros Leibensperger, Moshe Braner, and David Fielding. Transputer performance issues using the trollius operating system. In H. Zedan, editor, *Real-Time Systems with Transputers*, pages 35–49. IOS Press, 1990.
- [6] Y. Benzion and J. R. Rice. Earthquake failure sequences along a cellular fault zone in a 3-dimensional elastic solid containing asperity and nonasperity regions. *Journal of Geophysical Research Solid Earth*, 98(B8), Aug 10 1993.
- [7] W. Bitterlich and D. Ratte. Canadian biogenic emissions inventory system (canbeis). Course Project for Department of Geography, University of Ottawa, December 1992.

- [8] N. Boccara and K. Cheong. Critical behaviour of a probabilistic automata network sis model for the spread of an infectious disease in a population of moving individuals. *Journal of Physics a Mathematical and General*, 26(15), Aug 7 1993.
- [9] N. Boccara, K. O. Cheong, and M. Oram. a probabilistic automata network epidemic model with births and deaths exhibiting cyclic behaviour. *Journal of Physics a Mathematical and General*, 27(5), Mar 7 1994.
- [10] F. Bonfatti, G. Gadda, and P. D. Monari. Simulation of dynamic phenomena by cellular automata. *Computers & Graphics*, 18(6), Dec 1994.
- [11] G. D. Burns. Trollius: Early american transputer software. *Parallelogram*, (13), 1989.
- [12] G. D. Burns. *Alex-Trollius Command Reference*. Alex Informatique, 1992. Based in part on the 2.0 edition of Trollius User's Reference, dated July 5, 1990, Copyright 1990 The Ohio State University and Cornell Research Foundation.
- [13] G.D. Burns. *Alex-Trollius Tutorial Introduction in C*. Alex Informatique, 1992.
- [14] K. Clarke, J.A. Brass, and P.J. Riggan. A cellular automaton model of wildfire propagation and extinction. *Photogrammetric Engineering and Remote Sensing*, 60(11):1355-1367, 1994.
- [15] E. F. Codd. *Cellular Automata*. Academic Press, New York, 1968.
- [16] L.A. Crowl. How to measure present and compare parallel performance. *IEEE Parallel and Distributed Technology*, Spring 1994.
- [17] Frank Dehne, Claire Kenyon, and Andreas Fabri. Scalable and architecture independent parallel geometric algorithms with high probability optimal time. Technical report, School of Computer Science Carleton University, March 1994.
- [18] G. Engelen, R. White, I. Uljee, and P. Drazan. Using cellular automata for integrated modelling of socio-environmental systems. *Environmental Monitoring and Assessment*, 34(2), Jan 1995.

- [19] D. Engellhardt. Evaluation of characteristics of three programming paradigms supported by existing cm5 software. Obtained from the World Wide Web, 1993.
- [20] Hermannsson et al. Ec/dsim: a frontend and simulator for huge parallel systems. In *Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*, volume I: Architecture, pages 241–50.
- [21] Pombortsis et al. Pase: a performance analysis simulation environment. *Simulation Practice and Theory*, 2(1):43–59, 15 Sept 1994.
- [22] M. Gardner. The fantastic combinations of john conway's solitaire game of life. *Scientific American*, 223(10):120–123.
- [23] B. Grieger. Quaternary climatic fluctuations as a consequence of self-organized criticality. *Physica A*, 191(1-4), Dec. 15 1992.
- [24] Forestry Canada Fire Danger Group. Development and structure of the canadian forest fire behaviour prediction system. Technical report, Forestry Canada, 1992.
- [25] P. Brinch Hansen. Parallel cellular automata: A model program for computational science. *Concurrency: Practice and Experience*, 5(5):425–448, August 1993.
- [26] J. R. Henderson, I. G. Main, C. Maclean, and M. G. Norman. a fracture-mechanical cellular automaton model of seismicity. *Pure and Applied Geophysics*, 142(3-4), 1994.
- [27] F.C. Hoppensteadt. Mathematical aspects of population biology. In L.A. Steen, editor, *Mathematics Today: Twelve Informal Essays*, pages 297–320. Springer Verlag, 1978.
- [28] D. Hutchinson, L. Küttner, M. Lanthier, D. Nussbaum A. Maheshwari, D. Roytenberg, and J.-R. Sack. Parallel neighbourhood modeling. 1996.
- [29] D. Hutchinson, L. Küttner, M. Lanthier, D. Nussbaum A. Maheshwari, D. Roytenberg, and J.-R. Sack. Parallel neighbourhood modeling: Research summary. In *Proceedings of the Symposium on Parallel Algorithms and Applications*, pages 204–207, 1996.

- [30] D. Hutchinson, M. Lanthier, A. Maheshwari, D. Nussbaum, D. Roytenberg, and J.-R. Sack. Parallel neighbourhood modelling. Submitted to Conference, August, 1996.
- [31] R. M. Itami. Simulating spatial dynamics - cellular automata theory. *Landscape and Urban Planning*, 30(1-2), Oct 1994.
- [32] R.M. Itami. Cellular automata as a framework for dynamic simulations in geographic information systems. In *Proceedings of GIS/LIS'88*, volume 2, pages 590-597, 1988.
- [33] J. Jája. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, 1992.
- [34] F. Jeltsch and C. Wissel. Modelling dieback phenomena in natural forests. *Ecological Modelling*, 75(SEP), Sep. 1994.
- [35] C. MacDonald. Node view: A profiler/debugger for parallel computers using message passing. Master's thesis, Carleton University, 1996.
- [36] R. Miller and Q. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, 38(3), March 1989.
- [37] J.E. Neilson. *The Parasol 3.0 Reference Manual*. Carleton University, 1995.
- [38] Q. Ning, V. Van Dongen, and G. R. Gao. Automatic data and computation decomposition for distributed memory machines. *Proceedings of the 28-th Hawaii International Conference on System Sciences Wailea, Maui, Hawaii, January 1995*.
- [39] Greg Burns of Ohio State University. Personal communication, August 1995.
- [40] M. Phipps, J.Phipps, J.F. Whitfield, A. Ally, R.L. Somorjai, and S. A. Narang. Carcinogenic implications of the neighbourhood coherence principle (necp). *Mutational Hypotheses*, 31:289-301, 1990.

- [41] M.J. Phipps. From local to global: the lesson of cellular automata. In D. L. De Angelis and L.J. Gross, editors, *Individual based models and approaches in ecology: populations, communities and ecosystems*, 1992.
- [42] R. Raghavan, J. Kreulen, B. O'Krafka, S. Salamian, A. Saha, and N. Malik. Multiprocessor system verification through behavioral modeling and simulation. In *Proceedings of the Fourteenth Annual International Phoenix Conference on Computers and Communications*, pages 396–402. IEEE, 1995.
- [43] D. A. Rand. Measuring and characterizing spatial patterns, dynamics and chaos in spatially extended dynamical systems and ecologies. *Philosophical Transactions of the Royal Society of London Series a Physical Sciences and Engineering*, 348(1688), Sep 15 1994.
- [44] D. Roytenberg and J.-R. Sack. A simulator for the alex avx series ii parallel computer. In *Proceedings of the 10th Annual International Symposium on High Performance Computers*, June 1996.
- [45] David Roytenberg. *SAS Version 1.0 User Manual*. School of Computer Science, Carleton University, 1995.
- [46] A. Saarimaki. Bulk synchronous parallel: Practical experience with a model for parallel computing. Master's thesis, Carleton University, 1995.
- [47] K. Sato and Y. Iwasa. Modeling of wave regeneration in subalpine abies forests - population dynamics with spatial structure. *Ecology*, 74(5), Jul 1993.
- [48] W.R. Tobler. Cellular geography. In S. Gale and G. Olssen, editors, *Philosophy in Geography. Issues for Design and Implementation*, 1979.
- [49] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8), August 1990.

END

1 8 10 6 9 7

FIN