Programming Parallel Processors

Programming Parallel Processors

EDITED BY ***Robert G. Babb II***
Oregon Graduate Center

ADDISON-WESLEY PUBLISHING COMPANY, INC.
*Reading, Massachusetts • Menlo Park, California • New York
Don Mills, Ontario • Wokingham, England • Amsterdam • Bonn • Sydney
Singapore • Tokyo • Madrid • Bogotá • Santiago • San Juan*

This book is in the Addison-Wesley Series in Computer Science.

Library of Congress Cataloging-in-Publication Data

```
Programming Parallel Processors.

   Includes index.
   1. Parallel programming (Computer science)  I. Babb,
Robert G.
QA76.6.P75168  1988  004'.35                87-18833
ISBN 0-201-11721-5
```

Diagrams in this book were prepared by David C. DiNucci using gremlin and ditroff under Berkeley UNIX 4.3BSD on an Imagen 8/300 Laser Printer at Oregon Graduate Center, Beaverton, OR.
The text was typeset by ETP Services, Portland, Oregon, in Times Roman and Courier on a Linotronic 100 from troff source prepared by the editor.

ABCDEFGHIJ-DO-8987

Preface

This book reports practical experiences with programming commercially available scientific parallel processors. The intended audience is programmers, managers, and students in computer science and other disciplines with an interest in understanding the state of the art in software tools for programming the current generation of parallel processors. This record of our adventures should also prove of interest to the large number of software engineering researchers and system builders working actively today to develop better parallel programming languages and environments for the next generation of parallel processing computer systems.

This book developed from class projects in a graduate software engineering seminar that I taught at the Oregon Graduate Center in Spring 1986. Most of the students in the seminar had little or no experience with real parallel programming, although a few of the students had run parallel programs on the Department of Computer Science and Engineering's 32-node Intel iPSC Hypercube.

Although parallel computers are becoming increasingly available to the programming community today, the fraction of people (whether in industry or academia) who have actually run a parallel program is still small. Our experiences are probably representative of what any programmer might experience when first confronted with the brave new world of parallel programming. Some details reported in the chapters on programming the various machines are actually composites of some of the more interesting things that occurred during the course of our programming experiments, although they are written as if they all happened to the chapter author.

Since, by one recent count, there are currently over thirty companies worldwide that are building various flavors of parallel computers, we have been able to include only a small subset of machines in this compilation. The choice of machines was determined by the following criteria:

1. The machines had to be commercially available, rather than one-of-a-kind research testbeds.
2. The machines had to be accessible by the class. Generally, this meant that we needed either remote access via dial-up phone lines or travel support to gain

physical access to remote machines. Several machines (BBN Butterfly, Loral LDF-100, IBM 3090, and FPS T Series) were added to the list by class members after the class ended.

3. We wanted to include a wide range of machines, from supermini class (Sequent and Intel) to minisupercomputers (Alliant) to parallel vector supercomputers (CRAY, IBM).

4. We wanted experience with a variety of architectures and programming models, including both shared memory (Alliant, CRAY, IBM, and Sequent) and message-passing machines (BBN Butterfly, Intel iPSC, FPS T series, and Loral).

Since several of the machines that we worked on during the writing of this book were still quite new, the half-life for some of the implementation details discussed is quite short. In fact, it has been difficult for us to keep chapters up to date during the nine month period over which this book was written! However, some of the parallel programming environments have been fairly stable, and many of our more general observations about the nature of parallel programming should prove less perishable.

This book is not intended as a handbook on parallel computer architecture, although parallel architectural aspects are included for each machine where necessary to provide a basis for discussing various parallel programming issues.

In addition to the people who provided technical help with specific chapters, the editor would like to acknowledge the following people who read and commented on earlier drafts of the entire manuscript: Dan Hammerstrom, James Hardy, Ian Kaplan, Alan Karp, Richard Kieburtz, Kim Korner, and Alaine Warfield. The editor would also like to thank the reviewers of this work: Jack Dongarra, Argonne National Laboratory; Robert Hiromoto, Los Alamos National Laboratory; Harry Jordan, University of Colorado; and David Klappholz, Stevens Institute of Technology; for their suggestions and support for this project.

Beaverton, Oregon R.G.B. II

Disclaimer:

Although we include examples showing how parallel execution performance can be measured for most of the machines, and we describe various ways in which this performance data for our tiny Pi Program example could be interpreted, these results should not be interpreted as definitive, formal performance benchmarks. It would be a serious misuse of our data to draw general conclusions regarding the relative performance of the various processors based on our limited experiments with a single small parallel program.

Contributors

Timothy S. Axelrod, Lawrence Livermore National Laboratory

Timothy S. Axelrod received the B.S. degree in physics from California Institute of Technology in 1969, the M.S. degree in applied physics from Stanford University in 1971, and the Ph.D. degree in physics from the University of California, Santa Cruz, in 1980.

Currently he is Group Leader for Parallel Processing in the Computational Physics Division of Lawrence Livermore National Laboratory. His interests include radiative transfer and numerical modeling of supernovae, as well as performance issues in parallel computing. He has published widely in these and other areas.

Robert G. Babb II, Oregon Graduate Center

Robert Babb is an Associate Professor of Computer Science and Engineering at the Oregon Graduate Center. He received the B.S. degree in astrophysics and mathematics from the University of New Mexico in 1969. In 1974 he received the M.Math. degree in computer science from the University of Waterloo, Ontario, Canada, and the Ph.D. degree in electrical engineering and computer science from the University of New Mexico.

From 1974 to 1976 he was an Assistant Professor in the Computer Science and Statistics Department at California Polytechnic State University, San Luis Obispo. From 1976 to 1978, he was a Visiting Assistant Professor of Computer Science at New Mexico State University. From 1978 to 1982, he was a Software Research Engineer with Boeing Computer Services Company, Seattle, developing methods and tools for large-scale software engineering.

His current research interests center on the application of Large-Grain Data Flow (LGDF) methods to software engineering, data-driven parallel processing, and supercomputer system architecture.

Dr. Babb is a member of ACM and the IEEE Computer Society.

Contributors

Michael S. Beckerman, Tektronix

Michael Beckerman received the B.S. degree in computer science from Portland State University in 1984 and is working toward his M.S. in the Department of Computer Science and Engineering at the Oregon Graduate Center.

He is a Software Design Engineer working for Tektronix, Inc., and president of Dialectic Software Technologies, Inc., a software consulting firm.

His research interests include languages, specifications, programming environments, code generation, and Large-Grain Data Flow.

Frederica Darema-Rogers, IBM Hawthorne Research Laboratory

Frederica Darema-Rogers received the B.S. degree in physics and mathematics from the University of Athens, Greece, in 1969, the M.S. degree from the Illinois Institute of Technology in 1972, and the Ph.D. degree from the University of California at Davis in 1976, both in theoretical physics.

She was a Research Associate at the University of Pittsburg and Brookhaven National Laboratory and a Technical Staff Member at Schlumberger-Doll Research before joining IBM Research in 1982 as a Research Staff Member.

Her research interests are in the areas of parallel algorithms and techniques and tools for the development and performance analysis of parallel applications.

David C. DiNucci, Oregon Graduate Center

David DiNucci is a full-time graduate student in the Department of Computer Science and Engineering at the Oregon Graduate Center working towards an M.S. degree in computer science. He received the B.S. degree in computer science from Portland State University in 1983.

He has worked as a System Programmer at Portland State University and a Data Systems Coordinator for the Portland Public School District.

His current interests are in the fields of Large-Grain Data Flow and formal specifications.

Stuart W. Hawkinson, Floating Point Systems

Stuart Hawkinson received his B.S. degree in chemistry from Washington State University in 1965 and his Ph.D. in chemical physics from the University of Chicago in 1968. He performed post-doctoral research at Chicago under an NIH fellowship and was an NSF Post-Doctoral Investigator at Oak Ridge National Laboratory.

He is a Staff Scientist in the Product Definition Group at Floating Point Systems, where he has been actively involved in the development of parallel algorithms and architectures. Previously, he was a manager of the programming staff responsible for verification and performance enhancement of Floating Point Systems Math Libraries.

Before joining FPS, he was an Associate Professor of biochemistry and biophysics at the University of Tennessee, Knoxville, for ten years. He also held a Guest Scientist position at Oak Ridge National Laboratory in the Chemistry Division.

His current professional interests include numerical mathematics, parallel processing algorithms, and scientific computing applications.

Richard K. Helm, Floating Point Systems

Kent Helm is a graduate of Evergreen State College, Olympia, Washington, where he received his Bachelor's degree in computer science in 1982.

He is employed by Floating Point Systems as a Software Design Engineer specializing in parallel operating systems implementation.

Allan R. Larrabee, Boeing Computer Services

Allan Larrabee received a B.S. degree in biology and chemistry in 1957 from Bucknell University, Lewisburg, Pennsylvania. He received the Ph.D. degree in biochemistry with a minor in organic chemistry in 1962 from the Massachusetts Institute of Technology.

After two years in the Medical Service Corps, US Army, he did post-doctoral work at the National Institutes of Health, Bethesda, Maryland. In 1966 he became an Assistant Professor of Chemistry at the University of Oregon and in 1972 became an Associate Professor at Memphis State University, Memphis, Tennessee. He became a Full Professor at Memphis State in 1978.

He completed an M.S. degree in 1986 in the Department of Computer Science and Engineering at the Oregon Graduate Center. His thesis research was on adaptation of a large-scale computational chemistry program to the Intel iPSC Concurrent Computer. He is currently a Parallel Application Specialist for Boeing Computer Services, Bellevue, Washington.

James R. McGraw, Lawrence Livermore National Laboratory

Jim McGraw received the B.S. degree in computer science from Purdue University in 1972 and the Ph.D. degree in computer science from Cornell University in 1977. His thesis topic was "Language Features for Process Interaction and Access Control".

Dr. McGraw then became an Assistant Professor at the University of California, Davis, and a researcher for Lawrence Livermore National Laboratory (LLNL). During this time he became heavily involved in the design and use of applicative languages for multiprocessors. He is one of the principal designers of the SISAL language, which is now being implemented on a variety of multiprocessors by different research groups. Currently, he works for LLNL in the Computation Department. His administrative assignment is to organize, evaluate, and promote research projects in computer science for the laboratory.

His current research activities focus on the problem of writing highly parallel programs for multi-processor computers and automatically partitioning them for high system utilization.

Phillip C. Miller, Floating Point Systems

Phil Miller is a graduate of Purdue University, where he received his B.S.E.E. in 1978. He is currently pursuing an M.S. degree in computer science and engineering at the Oregon Graduate Center.

He is employed by Floating Point Systems as a Senior Software Engineer specializing in compiler design.

Contributors

Kurt B. Modahl, Oregon Graduate Center

Kurt Modahl is a part-time student pursuing an M.S. degree in computer science and engineering at the Oregon Graduate Center. He received his B.S. in natural science from the University of North Dakota in 1971.

He was a Research Associate at the Oregon Regional Primate Research Center from 1972 to 1982. From 1983 to 1984 he served as a Computer Consultant for Infotec Development, Inc.

His research interests are in parallel processing, program visualization, and object-oriented programming languages.

V. Alan Norton, IBM Yorktown Research Laboratory

Alan Norton received the B.A. degree from the University of Utah in 1968 and the Ph.D. degree from Princeton University in 1976, both in mathematics.

He was an Instructor at the University of Utah from 1976 to 1979 and an Assistant Professor at Hamilton College from 1979 to 1980. Currently he is a Research Staff Member at IBM, Yorktown Heights, New York, managing the parallel applications and architecture group of the Research Parallel Processing Project (RP3).

His research interests include the performance analysis and architecture of parallel computer systems, parallel algorithms, fractals, and computer graphics.

Douglas M. Pase, Oregon Graduate Center

Doug Pase received a B.S. degree in computer science and mathematics from Northern Arizona University, Flagstaff, Arizona. He is currently a full-time graduate student in the Department of Computer Science and Engineering at the Oregon Graduate Center.

He has designed and assisted in the design of compilers and parallel languages at Floating Point Systems and the Oregon Graduate Center.

His current research interests are in parallel language design, compilers, and advanced (parallel) computer architectures.

Keith E. Pennick, Boeing Computer Services

Keith Pennick received the B.S. degree in computer science from Washington State University in 1980. Currently he is a Systems Programmer for the High Speed Computing Center of Boeing Computer Services.

His primary interests are in parallel processing, networking, operating system design, and knowledge-based systems.

Gregory F. Pfister, IBM Research

Gregory Pfister received the S.B., S.M., and Ph.D. degrees in electrical engineering from MIT in 1967, 1969, and 1974, respectively.

He joined IBM in 1974, and from 1975 to 1976 was on the faculty of the Electrical Engineering and Computer Science Department of the University of California at Berkeley. In the IBM Research Division, he was Manager of Software Support for the Yorktown Simulation Engine (YSE) and is presently Principal Scientist of the RP3 project.

He has been elected to Eta Kappa Nu, Tau Beta Pi, and Sigma Xi, and is a senior member of the IEEE.

Charles E. St. John, Floating Point Systems

Chuck St. John is a graduate of Youngstown State University, Youngstown, Ohio, where he received his B.S.E.E. in 1977. He is currently pursuing an M.S. degree in electrical engineering from Oregon State University.

He is employed by Floating Point Systems as a Hardware Design Engineer specializing in the design of VLSI floating point arithmetic hardware. He is a member of IEEE and the IEEE Computer Society. His interests include parallel and high-performance architectures.

Stuart M. Stern, Boeing Computer Services Company

Stuart M. Stern received his B.S. degree in mathematics from Fairleigh Dickinson University in 1963.

He worked for The Boeing Company from 1964 to 1967, performing operating system support. From 1967 to 1972 he worked for Informatics, Inc., on several contracts, including IBM systems development, CBS News Presidential Election forecasting, and Air Force Intelligence graphics retrieval.

Mr. Stern was one of the founders of *CP/M Review* and *UNIX Review* magazines. He has been working for Boeing Computer Services since 1972. During this period, he was one of the principal designers of the original EXCHANGE ATM message switching system, designed a multitasking operating system for the MOSLER Corporation, and supported the AI Center's UNIX environment. Currently, he is working as an AI Specialist with Boeing Computer Services High Speed Computing Center.

Janice M. Stone, IBM Research

Janice Stone received the A.B. degree in mathematics from Duke University in 1962 and pursued graduate studies in mathematics at Georgetown University, and in logic and the philosophy of science at Stanford University.

She joined IBM research in 1984, where her research interests focus on parallel algorithms and tools for development and analysis of parallel programs.

Lise Storc, Oregon Graduate Center

Lise Storc is a part-time graduate student in the Department of Computer Science and Engineering at the Oregon Graduate Center working on her master's thesis. She received a B.S. in mathematics from the University of Texas at Austin in 1977 and attended graduate school in mathematics at the University of North Carolina at Chapel Hill.

She is currently employed as a Computer Scientist in the Medical Computing Research Laboratory at Emanuel Hospital and Health Center in Portland, Oregon, working on expert and graphics systems for a wide variety of medical applications.

Her current research interests are in parallel processing, medical expert systems, and the graphical display of complex mathematical objects.

Contents

1 Introduction

Robert G. Babb II

...WANTED for Hazardous Journey. Small wages, bitter cold, long months of complete darkness, constant danger, safe return doubtful. Honor and recognition in case of success.

—Ernest Shackleton[†]

Programming parallel processors is different. In 1984, when I ran my first parallel programs on the then brand-new Denelcor Heterogenous Element Processor (HEP) at Los Alamos National Laboratory, it quickly became apparent to me that parallel programming led to a higher "astonishment factor" than anything I had experienced in computing since my undergraduate days doing battle with PL/I.

The HEP had a very elegant and simple way of specifying synchronization operations in Fortran by reference to special *dollar-sign variables* (e.g., `$I`). The dollar-sign variables were shared via ordinary Fortran `COMMON`[‡] blocks between subroutines that could execute in parallel. Each dollar-sign variable had, in addition to its ordinary Fortran value (real or integer), a special bit that indicated whether the variable was *empty* or *full*. Attempting to assign a value to a full variable would cause a process to suspend until another process *emptied* the variable by reading a value from it. Similarly, a process that attempted to access the value of an empty variable (usually on the right side of an assignment statement) would be suspended automatically by efficient hardware

[†]From a newspaper advertisement for an Antarctic Expedition.

[‡]Throughout this book, including the reprinted material in the Appendices, we have used this font only for program text and machine values, and for characters typed on terminals. Words like *subroutine* (that are used by programmers as if they were normal English words) are generally not put in the special font unless they refer to a particular line of code containing the word `SUBROUTINE`. On the other hand, Fortran program elements such as `COMMON`, `IF`, and `DO` are usually set in this special font because their meaning in programs differs from their English meanings.

mechanisms, to come back to life after another process had written a value into the variable, with the side effect of *filling* it. To get more than one Fortran subroutine running, the `CREATE` statement, a parallel version of the ordinary Fortran `CALL`, allowed start up of separate, parallel *threads* of execution.

These two seemingly innocuous extensions to Fortran let loose the parallel genie on the world. One could now (in safe, old-fashioned Fortran, no less) create *semaphores*, *locks*, *processes* or *tasks*, *busy waits*, *barriers*, *critical sections*, and *monitors*. On the down side, programmers now had to deal with unpredictable and usually nonrepeatable situations of *deadlock*, *livelock*, *race conditions*, and *nondeterminism*. Suddenly, even very simple tasks, programmed by experienced programmers who were dedicated to the idea of making parallel programming a practical reality, seemed to lead inevitably to upsetting, unpredictable, and totally mystifying bugs. The difficulties we parallel pioneers experienced on the HEP seemed a lot worse than could be explained by the hardware and system software bugs that are common features with any very new computer system.

Since the coming (and, sad to say, passing) of the HEP, a large number of companies around the world have built a whole menagerie of commercial parallel machines. Some of them were built as special projects by established computer companies, but many have been built by startups that were able to convince venture capitalists that parallel processing was an idea whose time (and money) had come.

This book attempts to capture, at least at a tutorial level, the state of the art in programming commercially available scientific parallel computers. A number of other parallel machines that are specialized for such tasks as pattern matching [1] and signal processing have also appeared recently, but they are beyond the scope of this book. The machines we have included range in power from minicomputers to supercomputers. Their unifying feature is that they are all examples of commercially available scientific parallel machines that support *user-visible* parallel programming.

1.1 A BRIEF HISTORY OF PARALLEL PROCESSING

Parallel processing is not new. Operating systems have relied upon simulated and actual parallel operation of computers for at least twenty years. Hardware designers have dealt with the problems and rewards of parallelism at least since the days of von Neumann. In fact, early paper designs for what we know today as the von Neumann machine included consideration of a variety of parallel features. These parallel designs were rejected mainly because of the poor reliability of the components available for building machines. The designers' lack of experience in building any kind of computing engine also argued for adoption of the simplest possible design.

What *is* new is that computer manufacturers have begun to provide ways for application designers and programmers to control and exploit multiple CPUs directly to cooperate in solving a problem.

Some of the current confusion in the field of parallel programming is due to the wide variety of different computing cultures that have given us the terminology in common use among parallel programming afficionados. This also leads frequently to situations in which one basic concept can be described with three or four different words or phrases that have almost, but not quite, identical connotations.

An even larger discrepancy in terminology arises in the difference between *shared-memory* and *message-passing* machines. Each type of machine can simulate the other, but there seems to be a clear dividing line between the two camps, which of course is reflected in two overlapping but not identical sets of terms for related concepts.

1.2 PARALLEL PROCESSING TERMINOLOGY

When a particular instance of a code is executed on a machine, all of the work needed to execute that program is referred to as a single *task* or *process*. When a task executes on a multiprocessor, it can divide into several (possibly many) different and independent *threads* of execution. in the absence of other constraints, each of these threads of execution can execute simultaneously on different processors.

Each independent thread of execution is known as a *process*. It is often necessary for two or more processes to share information. For example, one process may compute some values that are used by another. If these values are stored in memory that is accessible to both processes, we describe it as being *shared data*. Shared data must always be accessed carefully to ensure correct program operation. We would not want one process writing the data while another is trying to read it.

A *critical region* refers to a section of code that must be executed with exclusive access to the shared data referenced within that code. A process preparing to enter a critical region may be delayed if any other process is currently executing inside a similar region. *Semaphores* or *locks* are one type of programming tool that can be used by programmers or compilers to implement critical regions safely.

On distributed memory machines (such as the Intel Hypercube), *messages* are used in much the same way that locks or semaphores are used on shared memory machines to *synchronize* computations. Of course, locks and messages are not mutually exclusive, since it is possible to conceive of hybrid machines which could make use of both kinds of synchronization simultaneously.

One way to coordinate multiple threads of computation periodically is to create a *barrier*. Several types of barriers have proven useful in scientific application programming. In one type, all processes in a group must arrive at a certain point in their code (the barrier) before any are allowed to proceed. A variant is to have a *single-thread* section following the barrier that any one (but only one) of the processes executes. A variant of this latter type of barrier is to allow only the *last* process to arrive to execute the single-thread critical region.

When a process attempts to *lock* (acquire) a lock, the act of getting the lock must be *atomic*. That is, only one of a set of processes should be capable of obtaining the lock. Processes that fail to get the lock can choose to *suspend* (when another suspended process is available to use the CPU), or can *busy wait* (sometimes called *spinning*) and keep trying repeatedly to obtain the lock. (This last strategy is obviously unattractive if it is possible that the lock will never be released!)

Deadlock refers to a situation in which each member of a group of processes is waiting for another member of the group to do something (typically, to unlock or release a lock). *Livelock* is more active but no more productive. It refers to a situation in which each member of a group of processes is busy signaling (passing messages) to other members of the group, but doing nothing to advance the progress of a computation.

In real life, things are sometimes very complicated. It is possible for some parts of a parallel program to be deadlocked, some parts to be livelocked, while another running process manages to compute the desired answer. It is even possible for sequential program bugs to masquerade as parallel errors [2].

1.3 SOFTWARE ENGINEERING ISSUES

Parallel programming can be even more frustrating than is regular programming. In addition to the usual software engineering problems common to all forms of program development, an additional set of problems must be avoided, and additional criteria must be met for a parallel program to be judged successful. Software engineering problems directly related to the introduction of parallelism include:

- Avoiding deadlock and livelock;
- Preventing unwanted race conditions;
- Avoiding the creation of too many parallel processes; and
- Detecting program termination (no longer a trivial matter!).

New evaluation criteria for parallel programs include:

- Program speedup versus number of processors;
- Size of synchronization overhead;
- Effect of problem size on speedup;
- Maximum number of processors that can be kept busy; and
- Determinism of program execution.

In addition, new software design issues arise, such as:

- What size program "chunks" should be used?
- How many parallel processes should be created?
- What form of process synchronization should be adopted?
- How should access to shared data be managed?
- How can deterministic program execution be guaranteed?
- How should processing tasks be subdivided to make the most effective use of available parallel hardware?

Debugging parallel programs is notoriously difficult. Race conditions can masquerade as program logic errors. When deadlock occurred on the HEP, for example, the addresses where the various parallel processes were hung (waiting for each other) could, after some effort, be determined. However, figuring out the sequence of events that got