Parallel Logic Programming

7-20

Parallel Logic Programming

Evan Tick

The MIT Press
Cambridge, Massachusetts
London, England

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Tick, Evan, 1959–
Parallel logic programming / Evan Tick.
p. cm. — (Logic programming)
Includes bibliographical references and index.
ISBN 0-262-20087-2
1. Parallel programming (Computer science) 2. Logic programming. I. Title. II. Series.
QA76.642.T53 1991
004'.35—dc20 91-2963
CIP

Logic Programming

Series Foreword

The logic programming approach to computing investigates the use of logic as a programming language and explores computational models based on controlled deduction.

The field of logic programming has seen a tremendous growth in the last several years, both in depth and scope. This growth is reflected in the number of articles, journals, theses, books, workshops, and conferences devoted to the subject. The MIT Press Series in Logic Programming was created to accommodate this development and to nurture it. It is dedicated to the publication of high-quality textbooks, monographs, collections, and proceedings in logic programming.

Ehud Shapiro
The Weizmann Institute of Science
Rehovot, Israel

Preface

People have been parallel programming for years on a wide variety of dual-processor mainframes, loosely-coupled distributed multiprocessors, array processors, vector machines, dataflow machines, shared-memory multiprocessors, etc. Yet in all these years, some of us found it hard to get excited about parallelism because there were few if any *high-class* parallel programming languages implemented on these machines. The lack of such languages was not completely due to lack of technology—the push to parallelize FORTRAN and other algorithmic languages was and still is very strong. What is a high-class language? It is a language that is *natural for programming*, all considerations of parallelism aside. We do not wish to trade away anything to gain performance improvement in exploiting parallelism: not declarativity, not clean semantics, not modularity, not correctness, not conciseness, nothing.

This book is an introduction to parallel logic programming languages, one (but not the only) family of high-class languages. The key development that inspired this book was the implementation of parallel logic programming languages on commercial shared-memory multiprocessors. Without the ability to measure actual performance tradeoffs, there is little point in writing a textbook about parallel processing as opposed to concurrent programming. Discussions of the beauty of concurrent semantics or the abundance of theoretical parallelism are only precursors to the acid test: the ability to achieve absolute performance improvement over the best sequential algorithms, as proved by timing statistics on real machines.

This book began at the Institute of New Generation Computer Technology (ICOT) as a series of performance benchmarks. The material was further developed concurrently with teaching the subject as a semester graduate course at the University of Tokyo during Spring 1989, and a quarter course at the University of Oregon during Summer 1990. The main purpose of these courses was to give the students experience in programming multiprocessors. Within two weeks students, many of whom had little experience with logic programming, were writing parallel programs and getting real speedups. This book should aid in developing a course wherein substantial parallel programming projects can be tackled by the students. Several such projects are suggested here. Prior knowledge of logic programming is not required; however, a strong programming background is desirable. The software systems used in this book, or ones similar to them, are available on a variety of multiprocessors.

It is time to make parallel programming as exciting as it should have been from the beginning.

Acknowledgments

Part of this book was written while I was a visiting researcher at the Institute of New Generation Computer Technology (ICOT) supported by a grant from the U.S. National Science Foundation. I thank Director Kazuhiro Fuchi and Dr. Sunichi Uchida of ICOT for their support of this work. I greatly enjoyed both working and socializing with the many researchers of ICOT's Fourth Research Laboratory. Especially helpful was Masatoshi Sato from the Oki Electric Industry Co. Ltd. who developed the real-parallel FGHC system on the Sequent multiprocessor. I apologize to him for taking the liberty of renaming his KL1PS system to Panda! Stimulating looks at the "big picture" with Manuel Hermenegildo, now at the University of Madrid, and Jim Crammond, now at Quintus Computer Systems Inc., were made possible by their separate visits to ICOT. I also thank Ewing Lusk and Ross Overbeek from the Argonne National Laboratory, and Andrzej Ciepielewski from the Swedish Institute of Computer Science, for their help with the Aurora system.

The remainder of this book was written at the University of Tokyo and at the University of Oregon. At Todai I was supported by a chair in Information Science endowed by the CSK Corporation. I thank my graduate students at Todai who overcame their public-university morass and worked quite hard. During summer 1990, Andrzej Ciepielewski visited the University of Oregon and used the text in his course. Andrzej and his students, notably Peter Adamson and Dae Yeon Park, were of invaluable assistance in correcting the text.

I thank Mark Korsloot from the Delft University of Technology, Kish Shen from Cambridge University, and Tim Lindholm from Quintus for their extensive comments on early drafts of this book. Sections 13.4 and Appendix B were greatly aided by help from Reem Bahgat from Imperial College, Daniel Dure from the French Ministry of Industry (on leave from Ecole Normale Supérieure), Rong Yang from Bristol University, and Kaoru Yoshida from ICOT. Lisa James, from the University of Oregon, did excellent copy editing of the book.

Contents

1 Introduction

"Two things, however, are impressed on novices: that all experiences are of equal spiritual significance (drudgery is divine); and that reasoning is futile. Zen holds that nobody can actually think himself into a state of enlightenment, still less depend on the logical arguments of others."
R.H.P. Mason and J.G. Caiger
A History of Japan
C.E. Tuttle Co., Inc. 1973

One of the most difficult problems with developing parallel processing systems is the job of parallel programming. By parallel programming we mean the programming of a single application to execute efficiently on multiple processors. The problem with parallel programming has been finessed, to some degree, by implementing sequential languages like FORTRAN on suitable multiprocessors, such as pipelined machines (e.g., CDC 6600), vector machines (e.g., CRAY-1), MIMD (multiple-instruction stream multiple-data stream) shared-memory machines (e.g., Alliant FX/8), and even MIMD pipelined machines (e.g., Denelcor HEP). The main reason for using sequential languages is the massive amount of code already written in those languages, as well as offloading the responsibility for "thinking in parallel" from the programmer to the compiler. The problem with sequential languages of the FORTRAN generation is that insufficient parallelism can be exploited from automatic translation alone [97].

From the genes of FORTRAN and ALGOL came families of *imperative* (procedural) parallel programming languages, e.g., Pascal Plus, Modula-2, Ada, and occam; and *applicative* parallel programming languages, e.g., SISAL, VAL, and ID. These languages use various methods of implementing parallel tasks, for example, the mutual exclusion of shared data updates and synchronization between tasks. The families were developed primarily to fill the need for design languages that could easily and clearly express parallelism.

While these families were being developed, and even earlier, other language designers were more concerned with designing languages that could easily and clearly express *the problem to be solved*. Examples of this latter family of languages are Lisp, Prolog, APL, and Smalltalk. Only recently have the two directions in language research met. Two major developments have been the implementation of parallel Lisp-like languages, e.g., Qlisp, MultiLisp, and MultiScheme, and parallel Prolog-like languages, e.g., Restricted AND-Parallel Prolog, OR-Parallel Prolog, Flat Guarded Horn Clauses (FGHC), Flat Concurrent Prolog (FCP), and Parlog.

This book is about how to program in two of these languages: OR-parallel Prolog and AND-parallel FGHC. Mastery of these two languages should easily facilitate the grasp of others. In addition, as Prolog offers a logical and clean approach to understanding programming in general, parallel Prolog-like languages (i.e., parallel Horn-clause logic-programming languages) offer a logical and clean approach to understanding the requirements of mutual exclusion and synchronization required by asynchronous parallel programming in general. Thus this book is an introduction both to logic programming and to parallel programming. Extensive programming examples are given, and their performance is analyzed using data collected on real shared-memory multiprocessor implementations. Performance data, although specific to these implementations and hardware hosts, is critical to understanding the tradeoffs in parallel programming. In many cases, we can abstract away the specific details of the system implementations and make concrete statements about efficient techniques for programming in these languages.

Logic programming, the paradigm of using first-order logic as the foundation of a programming language, is most popularly espoused in the form of Prolog. Prolog is a sequential language based on Horn-clause logic.[1] Prolog differs from procedural languages because it uses backtracking and unification, and is single assignment (within the scope of a clause). Prolog differs from functional languages in that Prolog has two-way unification, allowing a procedure to be executed in alternative modes. For example, a sorting program can be "run backwards" to produce permutations. Logic programming languages are distinct from almost all other languages in that logic programming languages naturally express a large number of different types of parallelism. The most renowned types are AND and OR parallelism. In general, AND-parallelism is the ability to execute two conjunctive tasks in parallel; OR-parallelism is the ability to execute two disjunctive tasks in parallel. In terms of logic programming, the task has the *granularity* of a goal execution, i.e., a procedure call and execution.

A goal execution is also called a reduction or logical inference.[2] The most promising types of AND-parallelism are *restricted* (sometimes called *independent*) and *stream*. Restricted-AND-parallelism avoids binding conflicts by guaranteeing, before spawning parallel goals, that the goals will not attempt to bind the same variable. Stream AND parallelism is the ability to 'stream' partially instantiated data structures from one conjunctive goal to the next. Binding conflicts are avoided by

[1]Objections that Prolog is *not* sequential should be saved until Section 1.2. To avoid ambiguity we sometimes write "sequential Prolog" meaning Prolog, as opposed to "AND-parallel Prolog" or "OR-parallel Prolog."

[2]Thus many systems claim performance figures in terms of KLIPS – thousands of logic inferences per second or KRPS – thousands of reductions per second. Beware however: not all logical inferences are equal because different goals may require different amounts of computation. Thus KLIPS is a very gross and often misleading metric.

suspending a goal when an input is unbound, and explicitly locking variables when binding them. In addition to these major types of parallelism, there are several other types, such as those described in Conery [33], Gregory [57], and Hermenegildo [62].

Although the theoretical importance of the presence of these various types of parallelism in logic programming languages is important, without practical implementations these results will not produce realistic speedups on multiprocessors. If programming becomes complex during the drive to efficiently implement these various types of parallelism, then sight of the original goal (to provide a *good* programming language that can be executed in parallel) will be lost. Thus logic programming language designers have been walking a fine line over the past several years — trying to exploit parallelism efficiently without weakening language semantics to the point of making programming impossible.

This book analyzes two vastly different approaches to this problem: OR-parallel Prolog and stream-AND-parallel FGHC (Flat Guarded Horn Clauses). Together, these languages represent the current state-of-the-art in parallel logic programming language design, in terms of both technology and programming methodologies.

On one hand, OR-parallel Prolog (also called "OR-Prolog" in this book) retains all the power of sequential Prolog, but exploits only the OR-parallel execution of nondeterminate clauses. OR-parallel execution involves running completely independent processes that cannot communicate with each other in any way. Any interrelated analysis of the solutions must be conducted on the group of solutions *after* the solutions have been constructed. Collection of independent solutions is performed in this book with the `findall` procedure. This builtin is an example of an *aggregation operator* that evaluates a goal for all its possible solutions [86].

On the other hand, FGHC sacrifices the ability to backtrack, i.e., to produce multiple solutions to nondeterminate problems, but exploits stream-AND-parallel execution of all goals. FGHC allows communication between processes, which permits the processes to collaborate during the search for solutions. However, the added burden of specifying communication often gives a program the characteristics of an intricate control structure. The program control structure is thus given the euphemism "the process reading," rather than the logical declarative reading. FGHC can be considered in some sense representative of a class of *committed-choice languages* including FCP [116], FLENG [89], Parlog [57, 35], Strand [49] and others.

This book is written as a programming primer, giving a progressive selection of annotated programs written in both OR-parallel Prolog and FGHC. The programs are used to expound certain programming techniques and pitfalls. In addition, performance timings are presented as evidence of why one type of programming methodology is better than another. These timings were collected from real parallel implementations of the languages on the Sequent Symmetry [95] and Encore Multimax [46] shared-memory multiprocessors. The programs given in this book progress from a trivial list-appending program, to variations of the classics, such as placing N queens on an $N \times N$ chessboard, to more advanced problems in semigroup theory and graph theory.

The importance of parallel processing is generally accepted in both the computer engineering community and the scientific (number-crunching) community. However, the importance of high-level programming languages is often neglected. Even within the so-called centers of logic programming research, little emphasis has been placed on developing sophisticated parallel debuggers or efficient optimizing compilers. One reason is that the research field is still young. The lack of tools puts greater emphasis on careful and efficient programming style. This book is meant as a guide to help "get it right the first time," because it may be some time before parallel programming environments approach those of sequential languages.

1.1 Some Pragmatics

Before eagerly jumping into the book, which is filled with possibly uncommon terms such as *clauses*, *guards*, *logical variables*, and *streams*, let us first consider (with tongue firmly in cheek) some examples of programming a multiprocessor in *the conventional way.* The three examples given, in increasing order of sophistication, speak for themselves.

1.1.1 Programming in UNIX

Figure 1.1 shows a manual page taken verbatim from the operating system of the Encore Multimax (UMAX 4.2) [46]. This particular page is concerned with spawning a parallel task from within a 'C' program. Especially amusing is the restriction concerning `printf`.

1.1.2 Programming With Monitors

Lusk *et al.* [77] give a clean, hierarchical approach to writing programs, for commercially available shared-memory multiprocessors, with *monitors*. Even though the 'C' language is used, the hierarchical use of macros keeps the programs uncluttered and portable. To explain the use of monitors, we review an example taken from Lusk *et al.*: a program which adds two vectors of integers (what could be simpler?). We present the 'C' program below without detailed explanation in order to give