FIFTH GENERATION COMPUTER SYSTEMS 1988

Proceedings of the International Conference on Fifth Generation Computer Systems 1988

Edited by
Institute for New Generation Computer Technology (ICOT)

Volume 3

OHM Springer-Verlag

FIFTH GENERATION COMPUTER SYSTEMS 1988

Proceedings of the International Conference on
Fifth Generation Computer Systems 1988
Tokyo, Japan
November 28-December 2, 1988

Edited by
Institute for New Generation
Computer Technology (ICOT)
Tokyo, Japan

Volume 3

OHM Springer-Verlag

FIFTH GENERATION COMPUTER SYSTEMS 1988
Proceedings of the International Conference on Fifth Generation Computer Systems 1988

Distribution
Sole distribution rights outside Japan granted to Springer-Verlag
All orders should be sent to the following addresses:

Japan:
OHMSHA LTD., 3-1 Kanda Nishiki-cho, Chiyoda-ku, Tokyo 101, Japan

North America:
Springer-Verlag NY Inc., 175 Fifth Avenue, New York, NY 10010

Rest of the world:
SPRINGER-VERLAG, Heidelberger Platz 3, 1000 Berlin 33, FRG

British Library Cataloguing in Publication Data
International Conference on Fifth Generation Computers: 1988
International Conference on Fifth Generation Computer Systems (FGCS '88).
1, Computer systems
I. Title
004
ISBN 3-540-19558-0

Library of Congress Cataloging-in-Publication Data
International Conference on Fifth Generation Computer systems (1988: Tokyo, Japan)
Proceedings of the International Conference on Fifth Generation Computer Systems 1988:
Nov. 28– Dec. 2, 1988, Tokyo Prince Hotel, Tokyo Japan/
Institute for New Generation Computer Technology.
p. cm.
Bibliography: p.
Includes indexes,
ISBN 0-387-19558-0 (U.S.)
1. Fifth generation computers — Congresses. I. Shin Sedai Konpyūta Gijutsu Kaihatsu Kikō (Japan) II. Title.
QA76.85.I58, 1988
004 — dc20 89-11366
CIP

ISBN-3-540-19558-0 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-19558-0 Springer-Verlag New York Berlin Heidelberg
ISBN 4-274-19558-0 OHMSHA, LTD.

Printed in Japan

TABLE OF CONTENTS

Volume 1

Volume 2

FOUNDATION

INVITED PAPER

SUBMITTED PAPERS

Functional Logic Programming

Theory of Parallel Computation

Formal Semantics

Program Analysis and Transformation (2)

Reasoning and Learning

SOFTWARE

Logic Programming Languages

Volume 3

ARCHITECTURE

SUBMITTED PAPERS

Parallel Prolog Systems

Parallel Architectures (1)

Parallel Architectures (2)

Implementation Techniques for Inference Machines

INVITED PAPER

SUBMITTED PAPERS

Parallel Inference Machines

ARCHITECTURE

PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1988,
edited by ICOT.

THE AURORA OR-PARALLEL PROLOG SYSTEM

Ewing Lusk
Ralph Butler
Terrence Disz
Robert Olson
Ross Overbeek
Rick Stevens
*Argonne**

David H. D. Warren
Alan Calderwood
Peter Szeredi†
Manchester‡

Seif Haridi
Per Brand
Mats Carlsson
Andrzej Ciepielewski
Bogumil Hausman
SICS§

ABSTRACT

Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors, developed as part of an informal research collaboration known as the "Gigalips Project". It currently runs on Sequent and Encore machines. It has been constructed by adapting Sicstus Prolog, an existing, portable, state-of-the-art, sequential Prolog system. The techniques for constructing a portable multiprocessor version follow those pioneered in a predecessor system, ANL-WAM. The SRI model was adopted as the means to extend the Sicstus Prolog engine for or-parallel operation. We describe the design and main implementation features of the current Aurora system, and present some preliminary experimental results. We conclude with our plans for the continued development of the system and an outline of future research directions.

1 INTRODUCTION

In the last few years, parallel computers have started to emerge commercially, and it seems likely that such machines will rapidly become the most cost-effective source of computing power. However, developing parallel algorithms is currently very difficult. This is a major obstacle to the widespread acceptance of parallel computers.

Logic programming, because of the parallelism *implicit* in the evaluation of logical expressions, in principle relieves the programmer of the burden of managing parallelism explicitly. Logic programming therefore offers the potential to make parallel computers no harder to program than sequential ones, and to allow software to be migrated transparently between sequential and parallel machines.

It only remains to determine whether a logic programming system coupled with suitable parallel hardware can realise this potential. The Aurora system is a first step towards this goal. Aurora is a prototype or-parallel implementation of the full Prolog language for shared-memory multiprocessors. It currently runs on Sequent and Encore machines. It has been developed as part of an informal research collaboration known as the "Gigalips Project".

The Aurora system has two purposes. Firstly, it is intended to be a research tool for gaining understanding of what is needed in a parallel logic programming system. In particular, it is a vehicle for making concrete, evaluating, and refining one (or more) parallel execution models. The intention is to evaluate the models not just on the present hardware, but with a view to possible future hardware (not necessarily based on shared physical memory).

Secondly, Aurora is intended to be a demonstration system, that will enable experience to be gained of running large applications in parallel. For this purpose, it is vital that the system should perform well on the present hardware, and that it should be a complete and practical system to use.

In order to support *real* applications efficiently and elegantly, it is necessary to implement a logic programming language that is at least as powerful and practical as Prolog. The simplest way to ensure this, and at the same time to make it easy to port existing Prolog applications and systems software, is to include full Prolog with its standard semantics as a true subset of the language. This we have taken some pains to achieve.

The bottom line for evaluating a parallel system is whether it is truly competitive with the best sequential systems. To achieve competitiveness, it is necessary to make a parallel logic programming system with a single processor execution speed as close as possible to state-of-the-art sequential Prolog systems, while allowing multiple processors to exploit parallelism with the minimum of overhead. This has been our goal in Aurora.

To summarise the objectives towards which Aurora is addressed, they are to obtain truly competitive performance on real applications by transparently exploiting parallelism in a logic programming language that in-

*Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

†*On leave from* SZKI, Donati u. 35-45, Budapest, Hungary

‡Department of Computer Science, University of Manchester, Manchester M13 9PL, U.K. *Now at:* Department of Computer Science, University of Bristol, Bristol BS8 1TR, U.K.

§Swedish Institute of Computer Science, Box 1263, S-164 28 Kista, Sweden

cludes Prolog as a true subset.

The main purpose of this paper is to describe the issues that must be confronted in or-parallel Prolog implementation and detail the decisions and compromises made in Aurora. We include benchmark tests that point out the strengths and weaknesses of some of these decisions. We conclude by describing some directions for further research.

2 BACKGROUND

In this section we describe the setting in which Aurora was developed and give a short history of the Gigalips Project.

2.1 Sequential Prolog Implementations

Prolog implementation entered a new era when the first compiler was introduced, for the DEC-10 [21]. The speed of this implementation, and the portability and availability of its descendant, C-Prolog, set a language standard, now usually referred to as the "Edinburgh Prolog". The DEC-10 compilation techniques led as well to a standard implementation strategy, usually called the WAM (Warren Abstract Machine) [22]. In a WAM-based implementation, Prolog source code is compiled into the machine language of a stack-based abstract machine. A portable emulator of this abstract machine (typically written in C) yields a fast, portable Prolog system, and a non-portable implementation of crucial parts of the emulator can increase speed still further. A parallel implementation of Prolog is achieved by parallelizing this emulator.

There are now many high-quality commercial and non-commercial Prolog systems based on the WAM. A parallel implementation can obtain considerable leverage by utilizing an existing high-quality implementation as its foundation. We use the Sicstus implementation [7], one of the fastest portable implementations.

Using a fast implementation is important for two reasons. Firstly, the single most important factor determining the speed of a parallel version is the speed of the underlying sequential implementation. Secondly, many research issues related purely to multiprocessing only become apparent in the presence of a fast sequential implementation. (Speedups are too easy to get when speed is too low.)

2.2 Multiprocessors

It is only in the last two years that multiprocessors have emerged from the computer science laboratories to become viable commercial products marketed worldwide. Startup companies like Sequent, Encore, and Alliant have made shared-memory multiprocessors commonplace in industry and universities alike. They are relatively inexpensive and provide a standard system environment (UNIX^{TM}) thus making them extremely popular as general-purpose computation servers. A similar revolution is happening with local-memory multiprocessors, sometimes called "multicomputers", but these are currently more specialized machines, despite their scalability advantages.

What the new breed of machines does *not* provide is a unified way of expressing and controlling parallelism. A variety of compiler directives and libraries are offered by the vendors, and while they do allow the programmer to write parallel programs for each machine, they provide neither syntactic nor conceptual portability. A number of researchers are developing tools to address these issues, but at a relatively low level (roughly the same level as the language they are embedded in, such as C or Fortran). A goal of the Gigalips Project is to determine whether it is feasible to propose logic programming as the vehicle for exploiting parallelism on these machines.

2.3 Or-Parallelism

As is well known, there are two main kinds of parallelism in logic programs, and-parallelism and or-parallelism. The issues raised in attempting to exploit the two kinds of parallelism are sufficiently different that most research efforts are focussing primarily on one or the other. Much early and current work has been directed towards and-parallelism, particularly within the context of "committed choice" languages (Parlog, Concurrent Prolog, Guarded Horn Clauses) [14, 20]. These languages exploit **dependent** and-parallelism, in which there may be be dependencies between and-parallel goals. Other work [11, 18] has been directed towards the important special case of **independent** and-parallelism, where and-parallel goals can be executed completely independently.

The committed choice languages have been viewed primarily as a means of expressing parallelism *explicitly*, by modelling communicating processes. In contrast, one of our main goals is to exploit parallelism *implicitly*, in a way that need have little impact on the programmer. This viewpoint has led us to take a rather different approach, and to focus in particular on or-parallelism.

There are several reasons for focussing on or-parallelism as a first step. Briefly, in the short term, or-parallelism seems easier and more productive te exploit transparently than and-parallelism. However, none of these reasons precludes integrating and-parallelism at a later stage, and indeed this is our ultimate intention.

- **Generality.** It is relatively straightforward to exploit or-parallelism without restricting the power of our logic programming language. In particular, we retain the ability we have in Prolog to generate all solutions to a goal.
- **Simplicity.** It is possible to exploit or-parallelism without requiring any extra programmer annotation or complex compile-time analysis.