GULP '90

Quinto Convegno sulla
Programmazione Logica

A cura di Annalisa Bossi

Padova, 6-8 Giugno 1990

Sala della Gran Guardia
Organizzato da:
Dipartimento di Matematica Pure ed Applicata - Università di Padova

Con il Patrocinio di:
Università di Padova
Comune di Padova - Servizi Speciﬁci e Manifestazioni

Coordinatore:
Annalisa Bensi (Università di Padova)

Comitato di Programma:
Pier Giorgio Basc (CSELT, Torino)
Alfredo Fum (Università di Catania)
Gilberto Fum (Università di Padova)
Gaetano Fum (Università di Pisa)
Alfredo Mannelli (Università di Torino)
Mauro Manselli (Università di Genova)
Paola Malta (Università di Bologna)
Daniele Matar (Università di Roma)
M. Vincenzo Marini (ELSAG, Genova)
Enrico Pagliero (Università di Padova)
Alberto Peressini (Università di Roma)
Cristina Sbog (D.S. Lapoa, Bologna)
Maria Simi (Università di Udine)
Silvio Valentin (Università di Milano)

Comitato organizzatore:
Nicola Cecchi (Università di Padova)
Sergio Delfi (Università di Padova)
Luigi Marangoni (Università di Padova)
Nino Tralli (LASSIS-CNRL Padova)

Con il Contributo di:
CNR
CERVEX S.p.A.
ICON s.r.l.
ELSAG S.p.A.
Systèmes & Management S. p. A.
ASI - Artificial Intelligence Software S.p.A.
Cassa di Risparmio di Padova e Venezia

Stampa: La Photograph, via del Santo, 48/6, Padova

INDICE

Prenessa
Prefazione

AMBIENTI E STRUMENTI
Performance of Logic+Functional Programming on a Distributed Memory Architecture
P.G. Bosco, C. Cecchi, C. Molino, M. Porta, G. Sofi
Verso un Supporto a Tempo di Esecuzione per Linguaggi Logici e Funzionali
M. Grapari, F. Saracce
The Epsilon KBMS: Environment and Applications
P. Cocia, S. Valeri
Compilazione ed Esecuzione di un Linguaggio Logico Distribuito
T. Castignetti, P. Ciancarini, M. Montanari

APPLICAZIONI
Comprensione di Descrizioni di Attività Economico-Produttive Espresse in Linguaggio Naturale
M. Fasolo, L. Garbi, N. Guarino
RASP: Resource Allocator for Software Projects
C. Bertazzi, M. G. Gatti, P. Giannotti
Logic Programming and Approximate Reasoning: Fuzzy Tools in Prolog Environments
E. Binaghi, D. Orban, A. Rampini
Integrazione di uno Strumento di Inferenza Probabilistica Basato sul Modello di Pearl nell’Ambiente LEAP
L. De Giovanni
Progetto e Sviluppo di ENpower, un Dimostratore Automatico di Teoremi Basato su EN-Strategy
F. Baj, M.P. Bonacina, M. Bruschi, A. Zani
VI

Logic Programming and Software Development Environments
V. Ambriola, P. Ciancarini

Strumenti per lo Studio di Reti di Petri sulla Programmazione Logica
A. Domenici

LINGUAGGI

LML: a Logic Based Language for Knowledge Manipulation
A. Brogi, P. Mancarella, D. Pedreschi, F. Turini

Estensioni di Ordine Superiore a Prolog Sono Necessarie
S. Costantini, P. Dell’Acqua, G.A. Lanzarone

Note sull’Uso e la Definizione di un Linguaggio per la Programmazione
Logica Strutturata
G. Rossi

Negazione come Inconsistenza e Meccanismi di Strutturazione Prolog
C. Bazzino, C. Charrere, B. Demo

OTTIMIZZAZIONI E TRASFORMAZIONI

Abstract Interpretation for Concurrent Logic Languages
C. Codognet, P. Codognet, M. Corsini

A Specialization of Bottom-up Abstract Interpretation for Type Inference
in Logic Programming
R. Barbuts, R. Giacobazzi

Optimizing Fully-Bound DATALOG Queries
N. Leone

Deriving Efficient Logic Programs for Unfolding Trees
M. Proietti, A. Petorossi

Fold/Unfold Transformations for Structured Logic Programming
M. Bugliesi, E. Lamma, P. Mello

SEMANTICA

SLD-Resolution Completeness without Lifting, without Switching
R. Sigal

VII

A Contribution to the Automated Treatment of Membership Theories
E. Omodeo, F. Parlamento, A. Policriti

Global, Local and Weak Graph Properties for Normal Logic Programs
A. Cortesi, G. Fité

A Theory for Modeling the Synchronization Mechanisms of Concurrent Logic
Languages
C. Palamidessi

La Semantica degli Aggiornamenti su Programmi Logici di Horn
L. Palopoli

Monotonicity in AE Theories: Preliminary Report
P. A. Bonatti

Costruzione di un Modello Minimale per una Procedura di Classificazione
Fuzzy
A. O. Arigoni, S. Nobilio

RELAZIONI INVITATE

Completeness Results for SLDNF
K. Kunen

Negation-as-Failure and Classical Negation
V. Lifschitz

Abstract Interpretation of Logic Programs: the Denotational Approach
K. Marriott, H. Sondergaard
Prefazione

Questo volume raccoglie i lavori presentati al Quinto Convegno sulla Programmazione Logica: GULP '90. Il Convegno è una occasione di incontro e di discussione tra tutti coloro che nel mondo accademico ed industriale svolgono attività di ricerca o sviluppano applicazioni nel campo della Programmazione Logica. Il Convegno, che si tiene a Padova dal 6 all'8 Giugno 1990, prevede quattro relazioni invitate e la presentazione di 27 lavori riguardanti le seguenti aree:
- ambienti e strumenti;
- applicazioni;
- extensioni e variazioni;
- ottimizzazioni e trasformazioni;
- semantica.

Desidero ringraziare il GULP, ed in particolare il suo Presidente Giorgio Levi, per la fiducia dimostrata nell’affidarmi il coordinamento del Convegno. Vorrei inoltre ringraziare tutti coloro che hanno contribuito alla sua organizzazione:
- i membri del Comitato di Programma ed i revisori per il lavoro di selezione dei lavori;
- Giuliana Dettori e Paola Mello per gli indispensabili consigli sull’organizzazione;
- i membri del Comitato Organizzatore: Nicoletta Cocco, Susi Dulli, Nino Trainito e Luigi Marcolongo, per la preziosa collaborazione;
- l’Università di Padova ed il Comune di Padova-Settore Spettacolo e Manifestazioni per il Patrocinio offerto;
- i relatori invitati, gli autori dei lavori e tutti i partecipanti.

Annalisa Bosci
Premessa


Il Convegno si propone principalmente le seguenti finalità:
- Servire da punto d'incontro per tutti i ricercatori ed utilizzatori di Programmazione Logica e permettere a molti di essi di esporre i loro risultati e problemi sia teorici che applicativi.
- Delineare lo stato dell'arte della ricerca nel settore attraverso relazioni tenute da ricercatori invitati.
- Avvicinare studenti e ricercatori alle principali tematiche della Programmazione Logica attraverso brevi corsi introduttivi.


L'ottimo livello delle relazioni programmate per la quinta edizione del Convegno che si svolge a Padova conferma la crescita di qualità ed il successo dell'iniziativa.

A nome del GULP ringrazio Annalisa Bossi e gli altri colleghi padovani per l'eccellenza organizzazione del Convegno.

Giorgio Levi
Presidente del GULP
AMBIENTI
E
STRUMENTI
Performance of Logic-Functional Programming on a distributed memory architecture

P.G. Bosco, C. Cecchi, C. Molso, M. Porta and G. Soffi
CSELT - via Reiss Romoli 274 - 10148 Torino

1. Introduction

The languages K-LEAF [Bosco87] and IDEAL [Bosco88] are proper extensions of Prolog to include functional features. They have been proved to be efficiently implementable through resolution and, consequently, the WAM technology [Bosco89b].

The high potential for parallelism exploitation in logic languages has been already shown by several parallel implementations, most of which on shared memory (bus-based) architectures. Our approach to parallelism was to investigate highly scalable structures, based on a large amount of nodes, each with its own local memory, where a notion of logically shared address space can be obtained through a suitable efficient interconnection network. We focused on OR-parallelism, but the class of application we were mainly interested in (real-time signal processing like image, speech, etc.), where, usually, blind OR-parallelism is not sufficient, requires some support for (multiple) best-first search. This implies (beyond some primitives for communication among OR-braches and for defining a notion of score) a dynamic process model, differently from the worker model of Aurora [Warren87] and PepYS [Baron88]. In accordance with these requirements we designed OP-Prolog [Giandomenico88,Molso89], an OR-parallel extension of Prolog conceived to be a powerful tool to program highly non-deterministic algorithms.

The problems of how to control the grainize of such processes (as well as the tasks in the worker model) is more evident on distributed architectures than on shared-memory ones. We devised a simple set of annotations for Prolog (different from the one available in Aurora) to distinguish sequential procedure from parallel ones and to guarantee the full sequential evaluation of a goal inside a parallel computation. The parallel version of K-LEAF is a proper extension of OP-Prolog, from which it inherited the annotation scheme [Bosco89a].

Regarding AND-parallelism (needed to support parallel reduction of strict functional terms) we restricted ourselves to the implementation of the mapping of AND to OR-parallelism [Bosco89a] like in [Carlsson88].

Along the paper we give an overview of the parallel abstract machine and its implementation on our distributed memory architecture PPES (sections 2 and 3) while section 4 is mainly devoted to a detailed analysis of the performance of OP-Prolog and parallel K-LEAF, introduced by a short discussion on the sequential performance which is important for a correct interpretation of results in the parallel environment.

2. Parallel K-WAM Abstract Machine

The Warren Abstract Machine (WAM) has currently become the undisputed state-of-the-art technique to efficiently implement sequential Prolog and parallel extensions of the language. Our K-WAM has been built on sequential WAM, preserving its basic mechanisms as much as possible and extending it along two almost disjoint directions [Bosco89a]: (a) the specific demands of OP-Prolog model [Giandomenico88], namely support of process execution and control of parallelism; (b) the requirements of K-LEAF model, which entails lazy evaluation of functional terms. The latter component has already been illustrated deeply in [Bosco89b]. The former aspect is very shortly sketched in this section; due to lack of room, we prefer to focus on a few major points (organisation of process run-time data structures, implementation of Cartus Stacks, Multiple Bindings, parallel call scheme), leaving out other nevertheless interesting issues (e.g. pure left termination and deallocation of shared blocks). The detailed description of the abstract machine of OP-Prolog is reported in [Molso89].
As well-known, the sequential WAM is a virtual machine provided with some dynamic areas: the Stack, the Heap, the Trail, the PDL, and some Static Registers. Three additional areas have been introduced in parallel WAM: Binding List (BL) and Binding Array (BA), with which a solution to the OR-conflict problem has been given (described later). A Control Stack, recordng Parallel Split Points, i.e. blocks of information related to the generation of new processes during a parallel call (the parallel counterpart of sequential choice points). Special attention has been paid to reduce the number of areas required. Therefore, for each WAM process the eight above-mentioned virtual areas have been compacted into four: (1) Control Stack, recording Status Registers, Split Points and the Trail, (2) Stack, including PDL, (3) Heap, including also BL elements; (4) BA.

A direct consequence of OP-Prolog eager process spawning is the need to support an environment of Cactus Stacks, into which the linear stacks of sequential implementation have to be transformed. To cope with the above requirement, the memory handling is based on a demand fixed-length block policy, namely the data memory has been partitioned into a collection of fixed-length blocks (typical satisfactory values ranging from 1K to 8Kbytes) which are delivered one at a time on demand as soon as a process needs to extend one of its areas (e.g. the Heap when allocating a new structure after a put_atom instruction). A serious problem arising from the block policy has been how to handle the so-called Seriality Tests, required in many situations during the dereference/holding algorithm (for instance, if older(NU) is true, i.e. if address x is older than (the contents of) B. register, the value bound to a location must be trashed). While in the sequential implementation these are very straightforward checks, consisting in a single address comparison operation, in the parallel case the strategy is a bit more complex but nevertheless satisfactorily efficient, based on the concept of Block Sequence Number (in integer qualifying the "age" of the block and written in its first cell, reserved to the purpose).

One of the main problems that a parallel implementation of Prolog must cope with is the OR-conflict, arising on variables which are still unbound when the computation forks at a split node. Multiple environments must be created so as to allow the alternative paths originated to assign different values to such variables.

A number of techniques have been proposed in literature, each one exhibiting its pros and cons. Anyway, the most interesting ones are centered around two basic concepts: Binding Array (BA) and Binding List (BL). Briefly, the former technique consists in transforming a reference to an unbound variable into an index of an array; multiple bindings can be simply created by assigning each process (or processor) its own BA copy. The BL is a generalization of the Trail of the sequential model: an assignment to a variable in a shared part of the stack can be done only indirectly, by adding a pair <var-address, value> to BL (obviously, the dereference algorithm must be modified accordingly and implies, in some cases, a sequential scanning of the list).

Our solution has consisted in an original combination of these two methods, appropriately tailored to the specific nature of OP-Prolog execution model and distributed-memory physical implementation. Each eager process creation and distributed-memory requirement would suggest to provide each virtual process with its own BA copy, so minimizing process interference and remote accesses; but indiscriminate duplication of BAs would be impractical, since BAs are potentially ever-growing structures, unless appropriate measures are taken. As a consequence, we have distinguished a special class of variables, so-called parallel local unbound, recognized at compile time (i.e. those local variables in a parallel clause that will be unbound before a parallel split), for which the BA method is used. Multiple bindings for all the other variables are built through BL (but notice that in most cases, e.g. when referencing private areas of Stack or Heap, owing to the close separation between OP-Prolog sequential and parallel components, multiple bindings are not needed and assignments can be done directly, like in sequential Prolog).

For each of the basic constructs of the language (parallel call, parentof, all-solutions, etc) a minimal number of new special WAM instruction have been provided along with an appropriate compilation scheme. For instance, the expansion of a (many-clause) parallel procedure

comprises: (a) one (or more) parclause instruction sequences; (b) code for individual clauses.

- first_parclause \$41
- par_clause \$42
- par_clause \$43
- spawn

The above scheme has been inspired to the sequential indexing scheme (try / retry / throw instructions). Additional shorter parclause sequences, indexed by a switch_on_term instruction, may be expanded by the compiler. Code for each individual clause consists of a head (plus guard) part and a body part, separated by a semicolon instruction.

The head execution phase, started by first_parclause, consists in the sequential execution of WAM instructions corresponding to all alternative clause heads (and guards) and is terminated by spawn. The first_parclause instruction allocates in the Control Stack an appropriate data structure, the Parallel Split Point, containing information required for memory deallocation upon process termination, most notably the number of new child processes. The execution of each head may fail or succeed (in the latter case a new_procyce stores information to be used later by spawn), in both cases new child processes are activated by spawn (each one provided with its own copy of Binding Array). The parent process overtakes execution of left-most branch, extending the parent stack blocks (namely, Stack, Heap and Control Stack), while new blocks are dynamically allocated for child processes, so originating the above-mentioned Cactus Stack structure.

3. Distributed implementation of K-LEAF on the PIPES architecture

As detailed in [Balboni87], the parallel architecture PIPES, developed at CSELT within ESPRIT Project n. 26, can be classified as a distributed memory MIMD (Multiple Instructions Multiple Data streams) processors, consisting in a 22-bit TRIO Transputer Array, where two distinct communication networks have been provided: a cut-through routing packet-switched Delta communication network, allowing very fast one-to-any communications, a bidirectional ring, obtained by simply interconnecting Transputer links, for not urgent large transfers of data.