Workshop on Parallel Implementation of Languages for Symbolic Computation

Editors: A. Ciepielewski
E. Tick

CIS-TR-90-15
July 30–31, 1990

Department of Computer and Information Science
University of Oregon

WORKSHOP ON PARALLEL IMPLEMENTATION OF LANGUAGES FOR SYMBOLIC COMPUTATION

University of Oregon, Eugene, July 30–31

Monday July 30

9:00 – 10:00	*Parallel Lisp Tutorial* Bob Kessler, University of Utah
10:00 – 10:30	coffee break
10:30 – 11:30	*Parallel Logic Programming Tutorial* Andrzej Ciepielewski, SICS and University of Oregon
11:30 – 1:00	lunch
1:00 – 1:30	*The Strand Language: Scientific Computing Meets Concurrent Logic Programming* Tim Mattson, Strand Inc.
1:30 – 2:00	*Evaluation of the KL1 Language System on the Multi-PSI* S. Onishi, Y. Matsumoto, K. Nakajima, K. Taki Institute of New Generation Computer Technology
2:00 – 2:30	*High-Level Abstractions for Symbolic Parallel Processing* K. Ho and P. N. Hilfinger, University of California at Berkeley
2:30 – 3:00	ice coffee break
3:00 – 3:30	*Meta-Parallelism in Prolog* Robert M. Keller, University of California at Davis
3:30 – 4:00	*Speculative Computation in Pure Prolog* Benjamin Yu, University of Toronto
4:00 – 4:30	*Parallel Execution of Speculative Computations* L. V. Kale and V. Saletore, University of Illinois at Urbana-Champaign

Tuesday July 31

9:00 – 9:30	*Implementation of Concurrent Utah Lisp* Mark Swanson, University of Utah
9:30 – 10:00	*Killing Processes in Qlisp* Ron Goldman, Lucid Inc.
10:00 – 10:30	coffee break
10:30 – 11:00	*An AND-Parallel Distributed Prolog Implementation* Ian Olthof, University of Calgary
11:00 – 11:30	*Task Switching in OPAL* John Conery, University of Oregon
11:30 – 1:00	lunch
1:00 – 1:30	*Andorra-I: An Implementation of the Basic Andorra Model* T. Beaumont, S. M. Raman, V. S. Costa, P. Szeredi, D. H. D. Warren, R. Yang, University of Bristol
1:30 – 2:00	*Performance Evaluation of a Dataflow/Von Neumann Hybrid Architecture for Parallel Execution of Logic Programs* S. Auwatanamongkol and P. Biswas, Southern Methodist University
2:00 – 2:30	*Side Effects in Prolog Programs* L. V. Kale, D. A. Padua, D. C. Sehr University of Illinois at Urbana-Champaign
2:30 – 3:00	ice coffee break
3:00 – 4:00	Shoot-out: *"What is the most important problem to be solved in implementing symbolic parallel processing?"*

The Strand Language; Scientific Computing meets Concurrent Logic Programming.

Timothy G. Mattson
Strand Software Technologies Inc.
15220 N.W. Greenbrier Parkway, Suite 350
Beaverton, OR 97006

Abstract

This paper introduces a relatively new member of the concurrent logic programming family known as Strand[1]. While Strand is being used for traditional applications in symbolic computing [2], it is breaking new ground for languages in its class by it heavy use in scientific computing [3]. This feat is accomplished through its interface to Fortran and C. Applications in protein structure determination and weather modeling are presented to support the claim that Strand is an effective tool for scientific computing.

1. Introduction

Parallel computing requires portable software development tools in order to realize its full potential. This is particularly the case in scientific computing for which application software frequently has a longer lifespan than the hardware upon which it runs.

Among the most elegant approaches for portable parallel programming, are those based on concurrent logic programming [4]. Unfortunately, the needs of scientific computing do not directly map onto the logic programming model.

The commercial implementation of the Strand language, STRAND88, has been extended into the domain of scientific computing by interfacing to Fortran and C. With this capability, the Strand language has been shown [3] to give numerically intensive programmers access to portable parallel computing.

This paper is a review of the Strand language and its use for the numerically intensive programs common to scientific computing. In particular, its use for weather modeling [3] and protein structure determinations [5] is described.

2. Strand as a Concurrent Logic Programming Language

Strand is a concurrent logic programming language. It builds on the research in Concurrent Prolog, Parlog, FCP and Guarded Horn Clauses to yield an efficient, high level language for programming concurrent computers[6].

Within the taxonomy of concurrent logic programming languages, Strand:

1. Is a static dataflow language (as is Parlog)
2. Uses single assignment variable binding as opposed to unification.
3. Has a totally flat process structure.

A tutorial introduction to strand is provided in the appendix. Its key features are:

Its computational model is based on extremely light weight processes.

forecasts over areas the size of the continental United States.The original sequential program is a 25,000 line Fortran program developed over a ten year period. For the sake of this study, the atmosphere was divided into 61x46km columns each of which were further divided into 16 horizontal layers. For this case, the original Fortran program poorly vectorized and ran at about 40 MFLOPS on a Cray X-MP.

The numerical method used in MM4 updates each grid point at time t+1 using values at a small number of spatial neighbors at time t. Hence, the parallel version of the program was produced by decomposing the three-dimensional data into subdomains, initially in one dimension and subsequently in two dimensions. Responsibility for each subdomain is allocated to a separate subgrid process. A central manager process collates the information generated by the subgrid processes. The computation on each subgrid process alternately updates the time for its own data and then exchanges boundary values with the neighboring subgrid processes.

The bulk of the development effort was concerned with getting the existing Cray code to run on a single processor of a Unix machine. Once this was achieved, parallel versions were developed rapidly.

The parallel codes duplicate most of the functionality of the existing sequential code. Hence,the concurrent component had to handle output of intermediate results, collection of global error estimates, etc. Despite this added complexity, the concurrent components of the codes totaled only 450 lines of Strand. Speedups of approximately ten over the original sequential code were achieved with 15 subgrid processes on a Sequent symmetry shared memory computer.

5. Application: Protein Structure Modeling

The three dimensional shape of a protein is a critical property for understanding the function of the protein. Full determination of protein's shape is a vast problem taxing the abilities of the fastest supercomputers. A valuable preliminary step is to predict where the protein folds into a particular shape known as an alpha helix.

Two Caltech chemists, Joe Bryngelson and John Hopfield [5] developed a sequential C program that learns how to predict alpha Helix locations. The program learns how to predict alpha helices for new proteins by carrying out a large nonlinear optimization calculation with known protein structures.

Stephen Taylor and Sam Southard [5] parallelized this sequential program using STRAND88 to express the parallel parts of the algorithm. 75% of the original C code was used in the final parallel program. Furthermore, Strand's management of the parallelism was very flexible allowing easy experimentation with different schemes for parallel execution.

Figure 1 presents a summary of the program's behavior on a Symult 2010 computer. The program displays linear speedup with up to 32 nodes. The drop off from linearity at 64 nodes is an artifact of the protein data set divided among 64 nodes. A larger data set would allow the program to show linear speed up beyond 32 nodes.

The logic required to manage parallelism inevitably carries a cost. As seen in figure 1, the cost incurred by the Strand/C program is small and is overcome by the time the second node is added.

[3] I. Foster and R. Overbeek. "Experiences with Bilingual Parallel Programing". Presentation at the Fifth Distributed Memory Computing Conference, 1990.

[4] Steve Taylor, Parallel Logic Programming Techniques, Prentice Hall, 1990.

[5] S. Southard Jr. "A Parallel Algorithm for Alpha Helix Prediction", Draft, March 12, 1990.

[6] Ian Foster and Stephen Taylor, "Strand: A Practical Parallel Programming Tool, Proceedings of the North American Conference on Logic Programming, MIT Press, 1989.

[7] Ian Foster, private communication.

[8] T. G. Mattson, "Strand Application Survey" , in preparation.

[9] T. G. Mattson, K. Steer. "The Seismic Signal Processing Workbench", SSTI Internal document, 1990.

Appendix: A Strand Tutorial

The key to any portable parallel processing language is its architecture independent view of the parallel computer. The Strand computational model is based on a pool of light weight processes representing the state of the computation.

A Strand computation cycle begins when the Strand Abstract Machine (SAM) removes a process from the pool. The SAM reduces this process into more primitive processes using the information from the Strand program. These processes are either placed back into the pool or, if they are so primitive as to be immediately executable, they are executed. These executable processes are referred to as *kernels*.

Strand kernels fall into two classes - user defined and language provided. Language provided kernels consist of assignment, scaler arithmetic, and other low level operations. User defined kernels are Fortran and C routines provided by the user.

In all cases, a process suspends until all of its required data is available. Hence, Strand can be viewed as a data flow language.

As mentioned earlier, Strand processes are reduced according to the rules from the Strand program. The Strand program consists of a collection of modules containing Strand procedures. A Strand procedure consists of a collection of clauses with the same name. The form of a clause is:

```
H :-  G1, G2, ... GM | B1, B2, ... BP.
```

H is the *head* of the clause. It contains the clause's name and the procedure arguments. G1 through GM are the optional guard kernels which provide tests of the procedure arguments. Finally, if the head matches the process being reduced, the guards all succeed, and all of the required data is available, then the optional body calls, B1 through BP, are activated. The body calls can be other Strand procedures or body kernels.

Strand variables are single assignment and are indicated by an initial uppercase letter. Processes share variables to communicate with each other.

input list is split into its head and tail during the matching. If the guard succeeds, `Status` is set to `ok` and the procedure terminates.

If the head matches but the guard from the first clause fails, the second clause is attempted. If this guard succeeds, the member procedure is recursively invoked on the tail of the list.

Finally, if the second argument of the process being reduced is an empty list, only the head of the third clause can successfully match. This indicates that the key was never found in the list and `Status` is set to the string `fail`.

This discussion implied that the Strand Abstract Machine considered the three clauses in sequential order. This is not required by the language. The programmer just provides for all cases that can arise during process reduction and does not have to worry about the order of the clauses.

Evaluation of the KL1 Language System on the Multi-PSI

Satoshi Onishi† Yukinori Matsumoto†
Katsuto Nakajima‡ Kazuo Taki†

† Institute for New Generation Computer Technology
‡ Mitsubishi Electric Corporation

Abstract

The Multi-PSI is a loosely coupled multiprocessor, which has been developed in the FGCS project for the purpose of providing a practical tool for research and development of parallel non-numeric software. It also served as a testbed for implementation of concurrent logic language KL1 on a loosely coupled multiprocessor.

This paper reports the cost measurement of intra- and inter-processor primitive operations in the systems. They show the basic performance of our distributively implemented concurrent language. Comments for the relationship to other language systems are included. Utilization of measurement results in parallel programming are discussed from the viewpoint of reducing the inter-processor communication overhead. Measurements of performance and communication overhead in benchmark programs are also shown.

1 Introduction

The Japanese fifth generation computer project has the target of building a highly parallel inference machine (PIM) on which we construct large-scale knowledge information processing systems. We have developed a prototype machine, the Multi-PSI system [Taki 88], that provides a practical tool for the research and development of parallel non-numeric software. It also serves as a testbed for an implementation of concurrent logic language KL1 [Chikayama 88] for a distributed memory architecture.

The Multi-PSI is a non-shared-memory multiprocessor, whose processing elements (PEs) are the same CPU hardware of the personal sequential inference (PSI) machine [Nakashima 87] (the microprogram is different). Up to 64 PEs are connected in an 8×8 two-dimensional mesh network with dynamic routing capability.

A distributed KL1 system was developed on the machine [Nakajima 89]. It is written in microprogram for execution efficiency. The design goal was to obtain overall high performance, taking into account garbage collection overhead, and to realize a distributed language system with a decentralized resource management mechanism for good scalability. The language system is easily expanded for a larger hardware than the current Multi-PSI with its 64 PEs.

This paper gives the cost measurement results of intra- and inter-PE primitive operations in the system, which decide the basic performance of a distributively implemented concurrent language system. Correspondence of those primitive operations to other language systems is commented on. The cost of inter-PE primitive operations gives a guideline for a programmer to control the grain size for better performance. Measurements of performance and communication overhead on benchmark programs are also shown with a discussion referring to the guideline mentioned above. Section 2 and 3 outline the

The table has pointers to goal stacks corresponding to physical priorities. The processor picks up the topmost goal of the highest non-empty goal stack and executes.

When a goal is executed, the guards of its defining clauses are tested. There are three cases:

(1) If one of them succeeds, the body part of the clause is executed;

(2) If all of them fail, a failure exception is raised;

(3) Otherwise, if none of them succeed and some of them block – that is, some of the input arguments are not sufficiently instantiated for guard test – the goal suspends on the variable(s) to be instantiated.

In case (3), a pointer to the suspended goal is written on the variable cells (the goal is said to be *hooked* onto the variables). When one of the variables becomes instantiated, the goal can be put back to the goal stack for scheduling.

The body part of a clause can contain body unification goals, body built-in predicate goals, and user-defined predicate goals. In the execution of the body part, a body unification is done *in-line*. The body built-in predicate goal is also executed in-line, except when one of its input arguments is uninstantiated. In this case, a goal which execute the built-in goal is created and is hooked onto the uninstantiated argument. Goal records are allocated for the user-defined body goals and pushed onto the top of the current goal stack (thus the scheduling is depth-first), except for the last one goal which is executed tail-recursively.

If the goal suspends, or succeeds but has no user-defined body goals, the next goal is picked up from the highest priority goal stack for execution.

The programmer can attach pragmas to user-defined body goals to specify execution priorities and where to move processor numbers. When a `@priority` pragma is attached to a goal, the goal is pushed onto the goal stack corresponding to the specified priority, not the previous one. When a `@processor` pragma is present, a `%throw` message is sent to the specified processor with goal information (code, arguments, priority, and so on). Only the surface level of the arguments are encoded into the message. Nested elements of lists and vectors are represented by external pointers. An external pointer is made up of a processor number and an index into the indirection table in that processor (called the *export table*). This indirection scheme was adopted, so that *local* garbage collection can be done on one processor without affecting external pointers in other processors pointing into that processor.

The value of an external pointer can be read by the `%read`/`%answer_value` protocol. A write to an external pointer is handled by the `%unify` protocol.

4 Intra-processor Operation Evaluation

4.1 Append Speed

An append (list concatenation) program is often used as a benchmark program for logic programming languages. An append program written in KL1 follows:

```
append([X|X1],Y,Z) :- true | Z=[X|Z1], append(X1,Y,Z1).
append(□,Y,Z) :- true | Z=Y.
```

The cost of one reduction (iteration) of the first clause is 39 steps of the micro instructions in the best case (no suspension, and so on), and the speed turns out 128 KRPS (Kilo Reduction Per Second) assuming no cache miss.

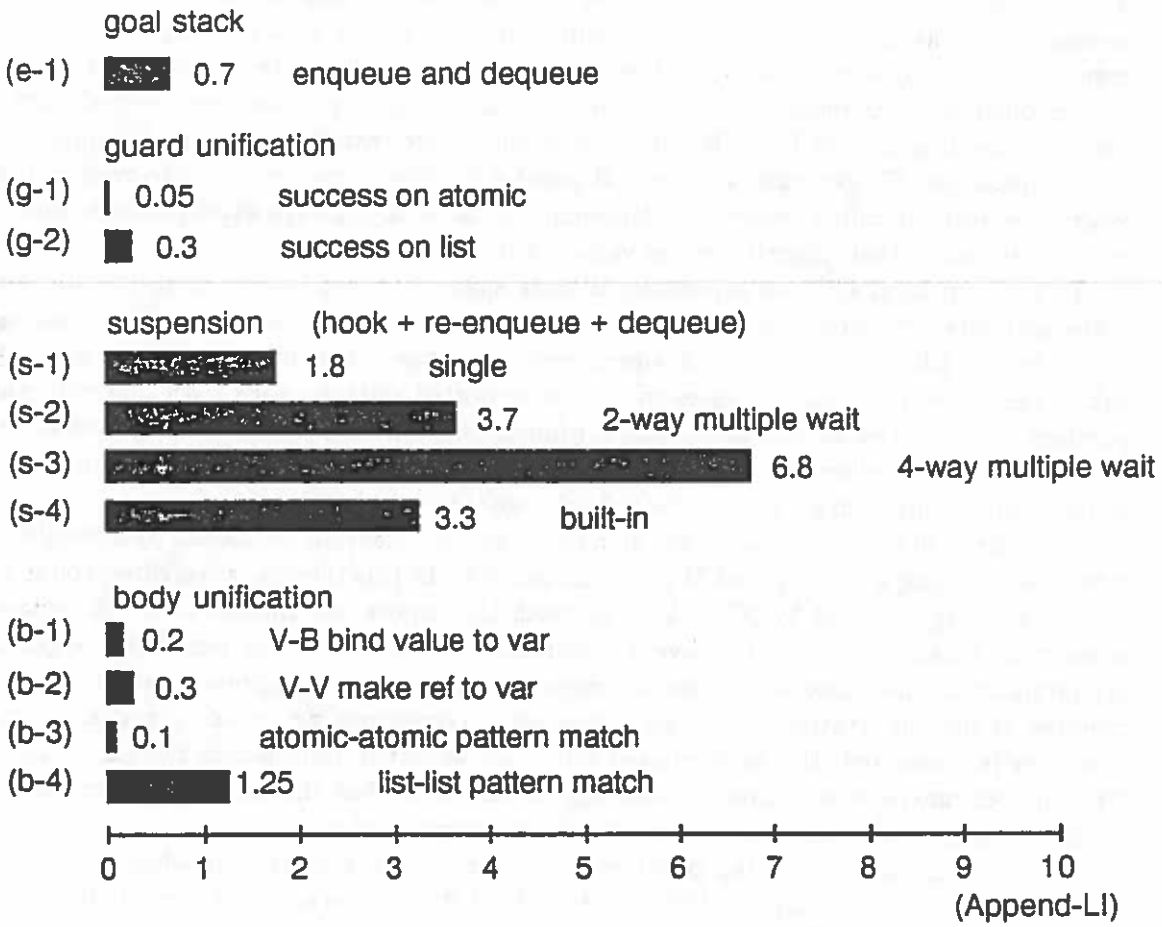


Figure 1: Cost of Typical Intra-PE Operation

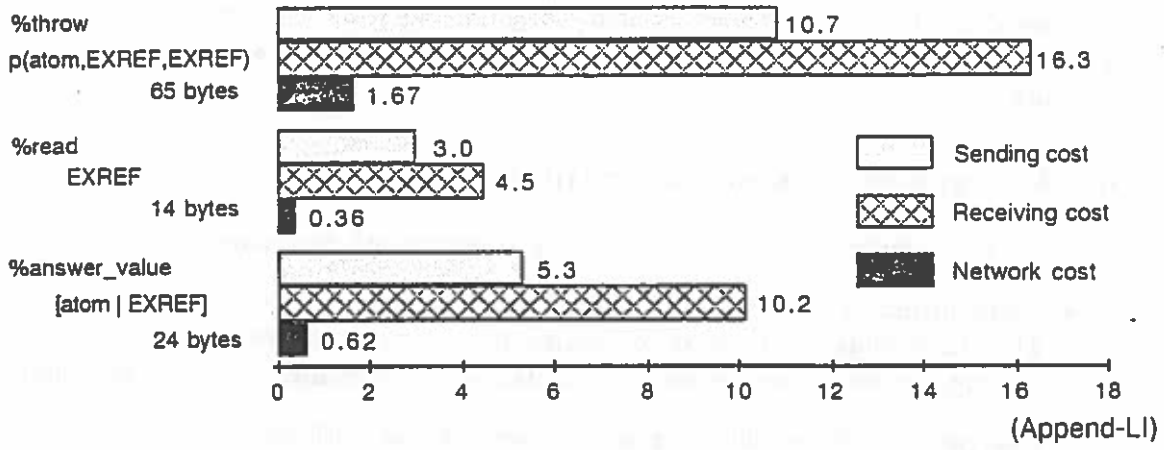


Figure 2: Cost of Typical Inter-PE Operation

the request. The returned data is a list whose CAR is an atomic data and the CDR is an external pointer.

The routing hardware has 5M bytes/sec of the bandwidth for transmitting the message. Compared with the network cost (hardware capability: 1.67/0.36/0.62 append-LI for 65/14/24-byte), the sending and receiving cost of the microprogram execution is quite large. It includes the cost of address translation, encoding and decoding messages, and distributed goal management and other resources management.

5.3 Discussion

Let's see the typical inter-PE operation in Figure 2, making an analogy to the discussion in section 3.3.

`%throw` operation corresponds to a process fork to a different processor. The cost of the `%throw` operation gives a guideline for a programmer to control the lower bound of the fork grain size thrown to a different processor.

`%read` and `%answer_value` operation correspond to a synchronization with a variable placed in a different processor. The cost of these operations gives a guideline to control lower bound of the synchronization grain size across a processor boundary.

Generally speaking, as the number of grains becomes larger, the load balancing becomes easier. In this case, the grain size tends to become smaller. A programmer divides his problem into smaller grains and distributes them among processors to get better load balance. But when the grain size becomes too small, the cost of `%throw`, or `%read` and `%answer_value` affect overall performance significantly. It means that a programmer has to keep the grain size larger than a certain size, which can be calculated based on the `%throw` cost or `%read` and `%answer_value` cost.

Let's see an example of deciding a lower bound of grain size. We assume a program in which only the synchronization grain size affects performance. Inter-PE synchronization cost, a summation of `%read` and `%answer_value` cost, is 23 append-LI. When a programmer accepts 50% performance down caused by those costs, the lower bound of the synchronization grain size becomes 23 append-LI. This means that interval of the inter-PE synchronization should be larger than 23 append-LI to keep the performance degradation below 50%.

There is an open problem whether the inter-PE operation cost shown in figure 2 is too large or not, compared with the intra-PE performance. We expect that the cost

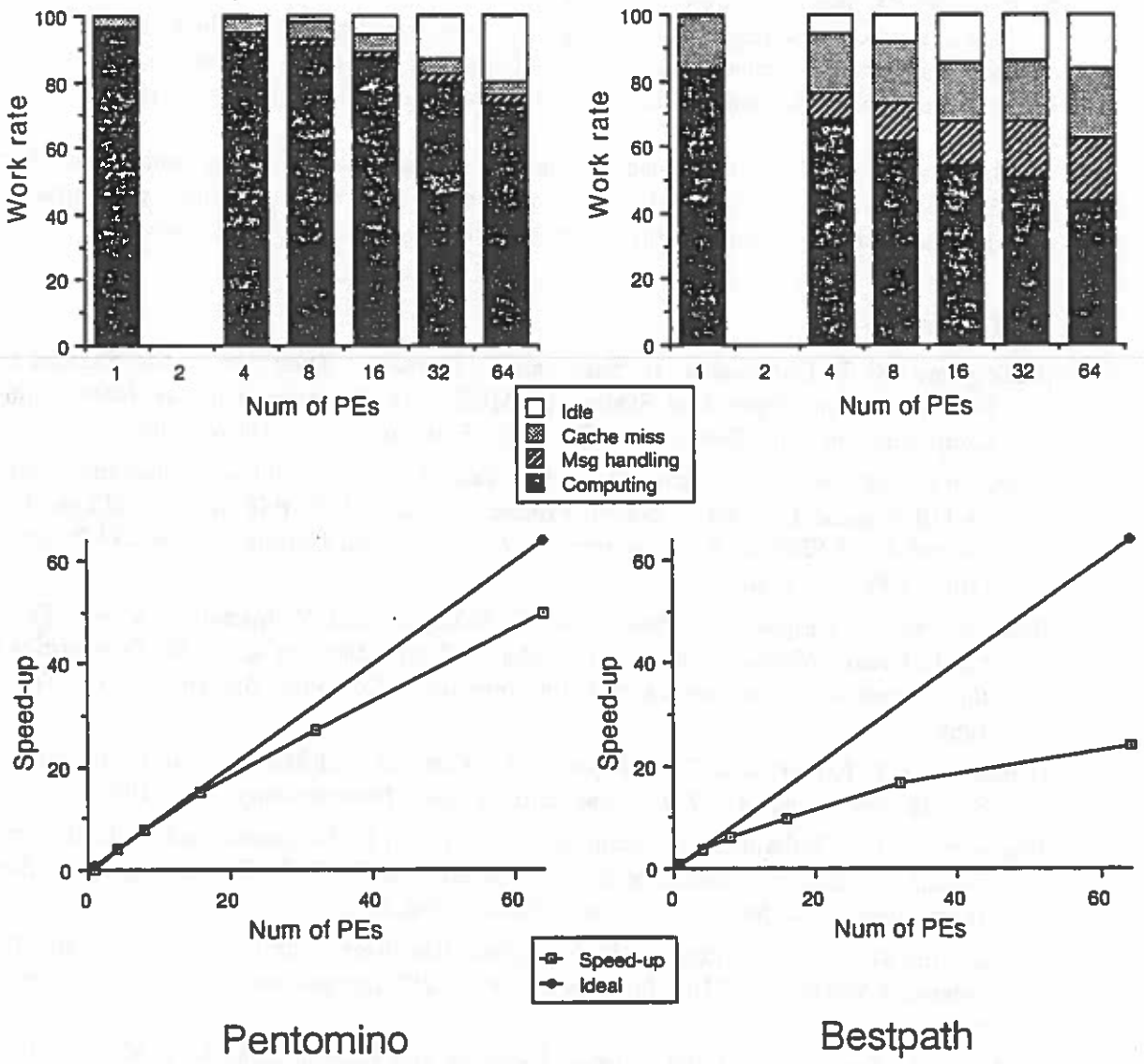


Figure 3: Runtime Analysis and Speed-up

High-level Abstractions for Symbolic Parallel Programming

Kinson Ho
Paul N. Hilfinger

Computer Science Division
University of California at Berkeley

July 19, 1990

Abstract

This paper describes a way of making symbolic parallel programming on shared-memory multiprocessors easier. It describes a toolbox of high-level programming abstractions and data structures for writing sequential and parallel programs with side effects. Programs written using this toolbox will run on uniprocessors and shared-memory multiprocessors without modification. Low-level parallel programming issues are handled by the runtime system, which provides good portable performance. The thesis of this research is that a small number of high-level abstractions are common to a large number of symbolic programs, and that these abstractions have efficient parallel implementations.

1 Introduction

We are interested in the efficient execution of symbolic Lisp programs on shared-memory (MIMD) multiprocessors. Lisp is being used for compilers, VLSI CAD programs for logic synthesis and simulation, expert systems, AI programs and rapid prototyping systems. These programs frequently take a substantial amount of time to execute, and can definitely benefit from speedups arising from the use of parallelism.

Techniques have been developed for the automatic detection of parallelism in FORTRAN programs. Parallelizing FORTRAN compilers usually attempt to schedule different iterations of a loop for execution in parallel on different processors, taking advantage of the regularity of control flow in FORTRAN programs. These

Control of Parallelism A parallel program performs optimally only if the problem granularity (or parallel-serial cutoff) is chosen such that the overhead of introducing parallelism (including process creation and scheduling) is justified by the resulting performance improvement. This optimal granularity is strongly implementation dependent, and even depends on the number of processors on a multiprocessor. Ad hoc schemes have been used to determine the optimal parallel-serial cutoff point for partitioning a problem. Measures such as the number of levels down a tree, the number of levels of recursion, the size of the task queue and the number of free processors are frequently used. These schemes do not generalize to other problems, and must be hand-tuned for each implementation separately.

Compatibility of Serial and Parallel Code The source code for the serial and parallel implementations of the same algorithm are often very different. This creates a program maintenance nightmare, especially if the serial implementation is still needed for portability and other reasons. For large programs this problem may be serious enough to justify *not* pursuing a parallel implementation, even when substantial performance improvements are expected.

3 Making Parallel Programming Easy

This research aims to make parallel programming easy for ordinary (sequential) programmers, which we propose to do by making parallel programming similar to sequential programming. In particular, we propose a toolbox of high-level abstractions for writing sequential *and* parallel programs. These abstractions represent common, time-consuming operations that offer good speedup potentials for parallel implementations. This section presents an overview of the toolbox. Details of the toolbox are given in Sections 4 and 5.

Ease-of-use and Performance Tradeoff This toolbox is intended for sequential programmers who need to utilize their multiprocessing hardware effectively, but are not interested or experienced in parallel programming. It is not intended for expert parallel programmers who are trying to achieve optimal performance on a given platform. Most of the parallel programming issues are hidden from the programmer. In particular, performance issues are handled by the runtime system. This toolbox approach is analogous to the performance-simplicity tradeoff between standard cell and full-custom styles of VLSI design.

The thesis of this research is that a small number of abstractions are common to the solutions of a large number of problems, and these abstractions have efficient parallel implementations.

```
Global:
    foreach variable Vj
        Vj := Unconstrained

    foreach constraint Ci
        Make-Task (Activate (Ci))


Termination condition:
    No more outstanding task (activation)

void Activate(Ci)
    foreach variable Vj connected to Ci
      Read value of Vj

    Compute consistent values for all (unbound) connected variables
        (new Vj s) of Ci

    foreach variable Vj connected to Ci

      Unify new Vj (value) with SHARED (current) Vj, update (atomic)

      if Vj is changed by unification
      then
          foreach constraint Cx (Cx != Ci) connected to variable Vj
              Make-Task (Activate (Cx))
```

Figure 1: Fixed-Point Algorithm for Consat

7 Related Work

This section describes different approaches for executing Lisp programs on multiprocessors.

Automatic Detection of Parallelism This approach is used to parallelize *dusty deck* FORTRAN programs by scheduling different iterations of a loop on different processors, making advantage of the regularity of control flow in FORTRAN programs. These techniques are much more difficult in Lisp because of the pervasive use of pointers, and the irregularity in data structures. Curare[8] and PARCEL[5] are examples of systems that reorganize sequential Lisp (Scheme) programs for execution on multiprocessors. In the absence of high-level knowledge about a program, these analyzers are often forced to make worst-case assumptions, limiting the amount of parallelism that can be extracted.

Parallel Lisp Dialects Most parallel Lisp dialects are extensions of Common Lisp or Scheme. They provide facilities for introducing multiple threads of control, and mechanisms for communication and synchronization. They differ as to whether threads share the same address space, whether they may be created dynamically, how synchronization is performed, and whether the communication is synchronous or asynchronous.

Multilisp: Multilisp[4] introduces the concept of `future`, which is a placeholder for the undetermined value of a computation. Futures are used to introduce parallelism between the computation of a value and its use. They can be inserted into sequential programs easily, but must be used with care in code with side effects. It is not clear *a priori* where they should be placed for optimal performance.

Qlisp: Qlisp[2] provides explicit ways for the programmer to control the degree of parallelism in a program. `Qlet` may be used to evaluate the variables of a `let` in parallel with the evaluation of the body. `Qlambda` may be used for client-server style interactions.

SPUR Lisp: SPUR Lisp supports explicit processes in a shared address space. Processes may communicate using shared variables, mailboxes (asynchronous communication using shared memory queues), and asynchronous signals. SPUR Lisp mailboxes provide a convenient interprocess communication mechanism not found in other versions of parallel Lisp.

Meta-Parallelism in Prolog*

(Extended Abstract)

Robert M. Keller
Division of Computer Science
University of California, Davis
rmkeller@ucdavis.edu

July 24, 1990

1 Overview

A variety of approaches toward parallelism in Prolog are under current investigation. Many of these deal with the detailed construction of efficient low-level machines for parallel Prolog execution (*cf.* [Borgwardt 84], [Hermenegildo 86], [Warren 87], [Lin and Kumar 88], [Lusk, et al. 89]). Although these have been carefully engineered for performance, they fail to expose at the programmer's level the exact mechanism being used to exploit parallelism, making it difficult for the user to understand and predict the performance of a particular Prolog program.

The direction of the current paper adopts the viewpoint that we wish to expose the execution model as much as possible, and even make it possible for the user to modify that execution model. This flexibility is viewed as important, since there are many semantic models for Prolog-like languages, offering a range of trade-offs with respect to logical completeness vs. efficiency. Thus, we present translation methods for taking source Prolog (having expected implicit parallelism) into Prolog endowed with simple primitives for explicit parallelism. One means of doing this is via a meta-

*This work was sponsored in part by a gift from Texas Instruments, Inc. The author thanks Quintus Computer Systems, Inc. for the use of Quintus Prolog, which was used to build the prototype implementations.

for the language.

3 The Knowledge-Server Semantic Model

The "knowledge server" model for parallel execution best represents our understanding of Prolog execution, and can be extended to incorporate parallelism. In this model, a Prolog program is viewed as a black box which accepts a query in the context of its knowledge base. The query will generally be a conjunctive term containing predicate constants and individual variables. Prolog tries to solve this query, where a solution is constituted by binding the variables in the goal to actual values. A solution then represents a term provable in the formal system represented by the Prolog program.

Since a query may have more than one solution, Prolog goes a step beyond and allows additional solutions to be requested. This can continue ad infinitum if the set of solutions is infinite. The knowledge server model shows this aspect of Prolog behavior by permitting the result of a query be a finite or infinite sequence of terms. The sequence can be empty, meaning that there is no solution to the query. This model is similar to that of [Debray and Mishra 88].

A denotational semantics of pure Prolog can be given based on the notion of knowledge server. In knowledge-server semantics, each clause of a procedure within a program defines a knowledge server. The semantics of a given procedure is defined recursively. Given an input goal, the sequence of results is the concatenation of the sequence of results from each clause individually. The sequence of results for a goal input to a single clause is defined as follows:

- If the goal does not unify with the head, then the result is the empty sequence.
- Proceed assuming that the goal is the most general instantiation of the original goal which unifies with the head of the clause and assuming that any bindings resulting from this unification have been substituted within the clause.
  - If the clause has an empty body, then the result is the sequence of one term, the (modified) goal itself.

parallel combinators which are interpreted at the meta level. We hasten to note that programming meta-interpreters within a fine-grain parallel implementation is not precluded.

- Executable specifications of various parallel processing interpreters provide performance comparisons with a common technology base.

5 Conclusion

We have presented a meta-parallel approach to parallel logic programming and discussed a variety of ways in which we feel it to be useful. These include: elucidation and understanding of various parallelism strategies for declarative and other languages, use of meta-interpreters as executives for coarse-grained parallel computation, and as a point of departure for new ways to express programs with clearly separated declarative and parallel control components.

6 References

[Borgwardt 84] P. Borgwardt. Parallel prolog using stack segments on shared-memory multiprocessors. IEEE First International Symposium on Logic Programming, 2-11 (February 1984).

[Bowen and Kowalski 82] K. A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. in K. Clark and S. Tarnlund (eds.), Logic programming, 153-172, Academic Press (1982).

[Ciepielewski and Haridi 83] A. Ciepielewski and S. Haridi. A formal model for OR-parallel execution of logic programs. Proceedings IFIP '83 (1983).

[Conery 87] J. S. Conery. Parallel execution of logic programs. Kluwer Academic Publishers (1987).

[Debray and Mishra 88] S.K. Debray and P. Mishra. Denotational and operational semantics for Prolog. J. Logic Programming, 5, 1, 61-91 (1988).

[DeGroot 84] D. DeGroot. Restricted AND-parallelism. International Conference on Fifth Generation Computer Systems, 471-478 (1984).

Speculative Computation in Pure Prolog

by

Benjamin Yu

Department of Computer Science
University of Toronto
Toronto, Ontario, Canada M5S 1A4

byu@csri.toronto.edu

1 Introduction

A number of research proposals have been made in the area of parallelizing Prolog along the lines of And, Or, and And/Or parallelism [1,3,5,7]. The two major obstacles in executing efficiently logic programs in parallel are the control of the proliferation of processes and the consistent binding of variables in each process. Considerable amount of overhead is spent on resolving these issues [12,2,11]. In this paper, we do not propose a new strategy to improve resolution of binding conflicts nor do we propose a new scheduler, but rather, we propose a parallel execution model for pure Prolog where binding conflicts may be avoided during execution and the number of processes created is controlled during the entire duration of execution. We turn to speculative computation for performance speedup.

2 Definition: Speculative Computation

Speculative computation is the precomputation of results or partial results of a subunit of a program before it is known whether the computed results or partial results may be necessary. It is a form of eager evaluation [20] and in the case where the result is useful and necessary for the computation of the program, it is hoped that it has been computed already when it is needed. An example is the statement in a program:

if a then b else c

During the execution of the program, either the code segment b or c will be executed, but before the actual execution, it is not known whether the evaluation of a will be *true* or *false*. Speculative computation of the program will attempt to precompute the code segments b and c as much as possible either during compile time or in parallel when the program begins execution. When the statement is actually executed, it is hoped that a performance gain can be realised over that when no precomputation of b and c is done. Speculative computation has been used to achieve speedup in parallel execution of processes in lisp. Osborne in [20] proposed a scheme to support speculative computation in Multilisp. Multilisp [9] features the use of the future construct where $(future\ X)$ (X is an arbitrary expression) creates a task to evaluate X and also creates an object which will eventually hold the value of X which is called a future. Parallelism using the future construct is possible by allowing processes to return a future as its value without waiting for it to resolve to the final value. This is implemented in the Mul-T parallel lisp system [17].

3 Speculative Parallelism in Prolog

In this section, we propose a framework for speculative computation in pure Prolog. According to the definition in the previous section, we need to 1) identify what is a subunit of a Prolog program and 2) the process of precomputation that may take place. A subunit of a Prolog program is the maximum subset of all the clauses that make up the program where the head of each clause in the subunit has the same predicate name. The precomputation of a subunit of a Prolog progam is a form of the unfolding rule proposed in [24] and an *unrecursing* rule which will be described in the next section. Before we do so, let us give an example of a program:

$$append([],X,X).$$
$$append([X|X0],Y,[X|Z]) \ :\!-\ append(X0,Y,Z).$$

$$member(X,[X|Y]).$$
$$member(X,[Y|Z]) \ :\!-\ member(X,Z).$$

with the query:

3.2 Model of Computation

Speculative computation results during execution when clauses are continually being unfolded and unrecursed. As mentioned earlier, the results or partial results from the speculative process may be unnecessary during the execution of the query. So the clauses being unfolded may be not used at all. However, we speculate that some of the clauses being unfolded (or at least one of them) will be used eventually. When an unfolded clause is needed, it is hoped that it would have been already generated.

Or parallel processes can be created for each alternative clause but fully unfolded and unrecursed clauses are usually mutually exclusive to each other. Thus efficient implementation and compilation of unfolded clauses may benefit more from the overhead cost incurred by Or parallelism. We will expand on this in section 4.

3.3 Use of Unfolding and Unrecursing Rules

The unrecursing rule is used to unfold predicates defined with one recursive clause. The partial-unrecursing rule is used to unfold predicates defined with more than one recursive clause. An example of the latter is the $split(H, L, U, B)$ predicate where the list L is split into two lists U and B such that all elements in U are greater than H and all elements in B are less than H.

$$
\begin{aligned}
&split(H, [H1|T1], [H1|U1], U2) \ :- \\
&\quad H1 < H,\ split(H, T1, U1, U2). \\
&split(H, [H1|T1], U1, [H1|U2]) \ :- \\
&\quad H1 > H,\ split(H, T1, U1, U2). \\
&split(\_, [], [], []).
\end{aligned}
$$

The two unrecursing rules are invoked to generate clauses unrecursed from depth 1 to depth N where N can be determined during the execution or specified by the user. The unfolding rule unfolds each predicate in the body by resolving each predicate and then unifying arguments with adjacent predicates and generates clauses with sometimes fewer predicates and less data dependency among them. See examples later in this section. Arithmetic and system predicates are not unfolded away.

The distinction in using the unrecursed and partial unrecursed rule is that unfolding a recursive predicate which is defined by more than one recursive clause can result in exponential number of clauses due to the explosive combination of the choices of clause to be unfolded. To generate the clause that may be needed to solve a goal predicate may require a long time. The partial unrecursed rule alleviates this problem by keeping the recursive predicate in the generated clause. This allows the process which succeeds in unifying with the generated clause to continue executing with the recursive predicate until the goal clause is solved. Clauses generated by the partial unrecursed rule are more efficient than the original clauses since they are partially unfolded and thus avoid additional clause invocation. It should be noted that unfolding a recursive predicate defined by only one recursive clause results in N clauses if the unrecursed rule is invoked for depth N. Hence, for predicates defined with more than one recursive clause, partial-unrecursed rule is used. Speed up will not be as significant as that results from using the unrecursed rule. However, in terms of space allocation and other implementation details, this may turn out to be more efficient for parallel execution.

It should also be noted that unfolded and unrecursed clauses do not replace the original clauses since, unless extra information is provided, it is unknown whether the unfolded and unrecursed clauses are "expanded" to the required depth for the entire duration of the execution of the program. We give here a few examples to illustrate the use of the unfolding and unrecursing rule.

Example 1

The append/3 predicate is unfolded into the following clauses:

$$
\begin{aligned}
&append([X], Y, [X|Y]). \\
&append([X, X0], Y, [X, X0|Y]). \\
&append([X, X0, X1], Y, [X, X0, X1|Y]). \\
&....
\end{aligned}
$$

Example 2

The predicate a checks if X is a member in the list Y, and if so, the list is processed by the predicate c.

$$
\begin{aligned}
&a(X, Y) \ :- \ member(X, Y),\ c(Y). \\
&\\
&member(X, [X|Y]). \\
&member(X, [Y|Z]) \ :- \ member(X, Z).
\end{aligned}
$$

$split(H, [H1, H2|T1], [H1, H2|U1], U2)$:-
$H1 < H,\ H2 < H, split(H, T1, U1, U2).$
$split(H, [H1, H2|T1], U1, [H1, H2|U2])$:-
$H1 > H,\ H2 > H, split(H, T1, U1, U2).$
$split(H, [H1, H2|T1], [H1|U1], [H2|U2])$:-
$H1 < H,\ H2 > H, split(H, T1, U1, U2).$
$split(H, [H1, H2|T1], [H2|U1], [H1|U2])$:-
$H1 > H,\ H2 < H, split(H, T1, U1, U2).$
$split(H, [H1, H2, H3|T1], [H1, H2, H3|U1], U2)$:-
$H1 < H,\ H2 < H, H3 < H, split(H, T1, U1, U2).$
$split(H, [H1, H2, H3|T1], U1, [H1, H2, H3|U2])$:-
$H1 > H,\ H2 > H, H3 > H, split(H, T1, U1, U2).$
....

The use of unrecursing rule eliminates the recursive call in *append* in Example 1, *member* in Example 2, *double* and *reverse* in Example 3. The unfolding rule unfolds *member* and unifies its arguments with *c* in Example 2. In example 3, arguments from the unfolded *double* and *reverse* are unified. Example 2 also shows that dependency between the predicate *member* and *c* is unfolded away.

3.4 Order of Unfolding

We now address the issues of when to unfold and which clauses should be unfolded in our model after clauses are unrecursed. Given a Prolog program, a Predicate Dependency Graph (PDG) is created. Each node represents a predicate and a directed path between node *a* and *b* (<*a*, *b*>) represents dependency of the variables used in predicate *a* on predicate *b*. An example is the following 4-queens program:

$queens([X1, X2, X3, X4])$:- $permutation([X1, X2, X3, X4], [1, 2, 3, 4]),$
$safe([X1, X2, X3, X4]).$

$permutation([], []).$
$permutation([X|Xs], Ls)$:- $delete(X, Ls, Rs),\ permutation(Xs, Rs).$

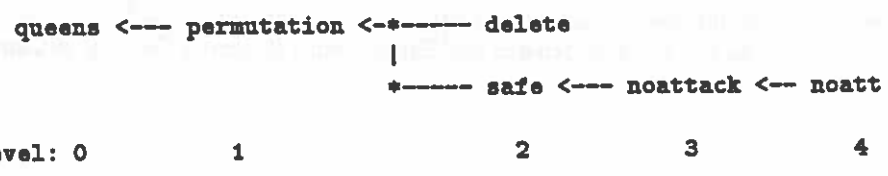
$delete(X, [X|Xs], Xs).$
$delete(X, [Y|Ys], [Y|Rs])$:- $delete(X, Ys, Rs).$

$safe([]).$
$safe([X|Xs])$:- $noattack(X, Xs),\ safe(Xs).$

$noattack(X, Xs)$:- $noatt(X, Xs, 1).$

$noatt(X, [], Nb).$
$noatt(X, [Y|Ys], Nb)$:- $T\ is\ Y - Nb,\ X \backslash== T,\ T1\ is\ Y + Nb,\ X \backslash== T1,$
$Nb1\ is\ Nb + 1,\ noatt(X, Ys, Nb1).$

The PDG generated from this program is:



Since the PDG is a directed graph, levels can be assigned to each predicate depending on its position in the graph. The graph also shows the order which the predicates will be executed. Levels higher than the second level will begin unfolding as soon as execution begins. More precisely, the highest level clause will start unfolding, followed by the next highest level, etc.. The result is unification of variables and elimination of subgoals. The first few levels of clauses can be compiled to generate unfolded clause of depth *n* before execution begins.

[3] Clark, K.L. and Gregory, S.; PARLOG: Parallel programming in logic; ACM Trans. Prog. Lang. Syst. 8,1; Jan 1986.

[4] Clocksin, W.F. and Alshawi, H.; A Method for Efficiently Executing Horn Clause Programs Using Multiple Processors; Technical Report, Computer Laboratory; University of Cambridge; 1986.

[5] Conery, J; The AND/OR Process Model for Parallel Interpretation of Logic Programs; PhD thesis; University of California, Irvine; 1983. Revised version appears in: Parallel Execution Of Logic Programs; Kluwer Academic Publishers; Boston; 1987

[6] Conery, J.; Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors; Proceeding of 1987 Symposium on Logic Programming; San Francisco, California; 1987.

[7] Degroot, D.; Restricted And-Parallelism; Proceeding of the International Conference on Fifth Generation Computer Systems; 1984.

[8] Dietrich, Suzanne Wagner; Extension Tables: Memo Relations in Logic Programming; IEEE Symposium on Logic Programming, 1987.

[9] Halstead, R.; Parallel Symbolic Computing; IEEE Computer 19:8; August 1986.

[10] Hausman, B. and Ciepielewski, A. and Haridi, S.; OR-parallel Prolog Made Efficient on Shared Memory Multiprocessors; Proceeding of 1987 Symposium on Logic Programming; San Francisco, California; 1987.

[11] Hermenegildo, M. and Nasr, R; Efficient Management of Backtracking in AND-Parallelism; Proceedings of the Third International Conference on Logic Programming; London, England; 1986.

[12] Janakiram, V., Agrawal, D., and Mehrotra, R.; A Randomized Parallel Backtracking Algorithm; IEEE Transaction on Computers; Vol 37, No 12; December 1988.

[13] Jefferson D. Virtual Time; TOPLAS; July 1985.

[14] Kalé, L.V.; 'Completeness' and 'Full Parallelism' of Parallel Logic Programming Schemes; Proceeding of 1987 Symposium on Logic Programming; San Francisco, California; 1987.

[15] Kalé, L.; The Reduce-Or Process Model for Parallel Evaluation of Logic Programs; Proceeding of the 4th International Conference on Logic Programming; 1987.

[16] Kowalski, Robert; Logic For Problem Solving; North-Holland, New York; 1979.

[17] Kranz, David and Halstead, Robert and Mohr, Eric; Mul-T: A High Performance Parallel Lisp; SigPlan 1989 Conference on Programming Language Design and Implementation.

[18] Li, P and Martin, A; The Sync Model: A Parallel Execution Method for Logic Programming; Proceedings of the 1986 Symposium of Logic Programming; Salt Lake City; Utah 1986.

[19] Lin, Zheng; Expected Performance of the Randomized Parallel Backtracking Method; North American Conference on Logic Programming; 1989.

[20] Osborne, Randy; Speculative Computation in Multilisp; Proceedings of U.S./Japan Workshop on Parallel Lisp; Sendai, Japan; June 5-8 1989.

[21] Ramkumar, Balkrishna and Kalé, Laxmikant; Compiled Execution of the Reduce-Or Process Model on Multiprocessors; North American Conference on Logic Programming; Cleveland, Ohio; 1989.

[22] Shapiro, E.Y.; A Subset of Concurrent PROLOG and its Interpreter; Technical Report TR-003, ICOT; Tokyo; 1983.

[23] Shapiro, Ehud; An Or-Parallel Execution Algorithm for Prolog and its FCP Implementation; Proceeding of the 1987 International Conference on Logic Programming; Melbourne, Australia; 1987.

[24] Tamaki, H. and Sato, T.; Unfold/fold Transformation of Logic Programs; Proceedings of 2nd International Conference on Logic Programming; Uppsala; 1984.

[25] Warren, D.H.D.; An Abstract Prolog Instruction Set; Technical Note 309; SRI International, AI Center, Computer Science and Technology Division; 1983.

Implementation of Concurrent Utah Lisp

Mark R. Swanson*
Center for Software Science
Department of Computer Science
University of Utah
Salt Lake City, Utah 84112

July 23, 1990

Abstract

This paper describes the implementation of Concurrent Utah Lisp, a language supporting distributed symbolic computation. Three different implementations of the language have been completed: one running on the BBN GP1000, another running on networked workstations, and a third running on the Mayfly, a distributed memory, twisted-torus communication topology, parallel processor under development at the Hewlett-Packard Research Laboratories. All of the implementations of the language have been heavily influenced by the Mayfly architecture, as will be shown.

1 Introduction

Concurrent Utah Lisp is an evolution of the Lisp language to support parallelism with tight coupling of control and data. The language model was in large measure motivated by the Mayfly processor, a descendent of the FAIM-1 Symbolic Multiprocessing System[DR85].

The intent in developing Concurrent Utah Lisp (CUL) has been to provide an efficient concurrent Lisp for distributed memory multiprocessors, in particular for the Mayfly architecture. The approach adopted has been to minimize the addition of new syntax and

*Work supported by Hewlett-Packard Research Labs - Palo Alto.

between processors. For a variety of complexity and performance reasons, we have chosen to copy data between processors. Copying that is simply an artifact of the placement of interacting objects would potentially yield different results with different placement strategies. Hence we chose to impose copying semantics at the language level.

Architectural Support – The decision to avoid implementing external data pointers was based in part on the expectation that concurrent Lisp would always be run on general-purpose processors (or at least not on special-purpose processors with support for runtime detection and resolution of external references). It was expected, however, that support *external to the processor* would be available to absorb some of the cost of communications.

Although we have introduced copying semantics at the inter-object level, CUL is not intended to be a functional language. The objects themselves can be shared, albeit in a mutually exclusive manner. This sharing is accomplished through the exported interface, which in CUL is defined in terms of closures. Thus, when a closure is transmitted from one domain to another, it cannot be copied, since this would be tantamount to copying the entire object and would preclude sharing of objects. Rather, the closure is transformed into a gateway, which is an executable object that implements monitor-like access to the shared closure.

1.2 Global Values.

Another major divergence from standard Lisp semantics involves CUL's treatment of global values. The language supports only single assignment of global values. Current implementations employ broadcast of these values to replicated symbol tables on each of the physical processors. The language also specifies that global values may not be destructively modified. The interactions of such modifications with copying semantics and with the distribution of such modifications would require so much mechanism as to make any use of globals prohibitively expensive.

each PE contains a communications processor (the Post Office)[SRD86], a dual ported data cache, twin instruction caches, two uni-directional hardware queues, and a set of context caches.

All message overhead is absorbed by the MP (except for the context switch required to move a task from the EP to the MP). This includes copying the message out of "Lisp space" into a message, packetization, and transmission for sending messages, and the analogous functions for receiving messages. The MP also creates and destroys tasks and performs scheduling duties.

The EP performs only application evaluation; whenever a message must be sent or a synchronization action performed, the EP switches out of the current application task and adds it to a hardware queue serviced by the MP. The EP then begins evaluating the next available task, supplied to it from the MP by means of a second hardware queue. To minimize context switch time, each task has associated with it one of the context cache blocks (a special purpose memory). Task private variables in the CUL implementation are assigned slots in the context cache; thus a context can be switched in just a few cycles by changing context caches. Examples of context cache-resident values are the heap allocation pointers. The context blocks are also used for passing arguments and results between the EP and MP.

The data cache consists of four sets. Two sets are handled as "private" to a particular processor (one each for the EP and the MP). A "private" set is not accessible to the other processor. Both these private sets respond only to the (same) particular portion of the address space reserved for "task private" data (e.g., stacks). The other two sets are managed as a 2-way, set associative cache.

The Mayfly architecture is intended to achieve high utilization and efficiency. It attempts to achieve this in three ways:

- by overlapping message overhead with evaluation of application code;
- by providing fast context switching to minimize required task granularity;
- by supplying sufficient interprocessor message bandwidth to prevent PE starvation.

2.3 Memory Management

The choice of copying semantics has had mixed results in its effect on memory managment for CUL. The encapsulation of data within domains provides the opportunity for very fast garbage collection within the "mini-heaps" thus created. This effect is enhanced by the encapsulation of all global data within a distinguished domain: the symbol table ceases to be a root for tracing in the process of collection for all other domains, dramatically decreasing the amount of work involved in a collection. Encapsulation also ensures that garbage collections need not be system-wide events, avoiding one requirement for global synchronization.

A further advantage to disjoint heaps accrues during the process of "lispifying" messages (copying them into the Lisp heap). The MP can gain exclusive access to the target heap; No global garbage collections exist to interfere (or require synchronization) and tasks running in other domains are free to execute (and allocate and garbage collect) on the EP, concurrently with the MP's copying activities.

Offsetting the benefits of encapsulation, though, comes a need for mechanisms to support export of placeholders and "gateways" (closures). This mechanism entails use of import and export objects and tables of indirection pointers to export objects. Together, these make it possible for tasks on one node to refer to objects existing on another node. These objects are managed using reference counts. The possibility exists that circular refences can result in persistence of unreachable objects. In addition, the maintenance of these reference counts does entail communicating changes in the counts to the owners (possibly on other nodes) of the referenced objects.

Finally, another level of storage management is introduced to handle explicitly allocated/deallocated objects. Currently a very simple buddy system (implemented at the Lisp level) is used. The objects thus managed have included the space used for the mini heaps, stacks and descriptor for tasks, export objects, and message buffers. Allocating all objects from a single space has the advantage of avoiding fixed and possibly unbalanced dedication of space to any particular use. The micro kernel can invoke the buddy system allocators deallocators, though the cost is rather high because of the differing function calling and register use conventions of Lisp and C. For this reason, some micro kernel allocations now

Parallel Execution of Speculative Computations*

L. V. Kalé and Vikram A. Saletore†
Department of Computer Science,
University Of Illinois at Urbana–Champaign,
Urbana, Illinois 61801

Abstract

Speculative parallelism arises frequently in many symbolic computations. It is the parallelism between two sub-computations, only one of which may be necessary to complete the top level computation. Many Artificial Intelligence computations involve problems for which many solutions are possible, and any one solution is adequate. We present an overview of our work on two different domains - state-space search and Logic Programming - that involve speculative parallelism.

1 Introduction

Consider the parallel execution of a heuristic search computation, where one is interested in any one of the possibly many solutions. Although searching alternate branches of the search tree may seem to be an obvious and highly parallel approach, it turns out to be quite tricky to get *consistent* speedups. We have developed a technique that uses priorities that ensures consistent (from run to run) speedups which increase *monotonically* with the number of processors (i.e. without anomalies). This parallel scheme was implemented using the *Chare-Kernel* parallel programming system [3], and shown to yield excellent speedups. The scheme was further extended to apply to the iterative deepening A* (IDA*) algorithm [7].

Substantially different extensions were needed to apply this technique in a process based OR parallel execution of Logic Programs [2], due to the way the OR tree gets mapped onto the process tree. We will summarise these techniques. Adding AND parallel execution, and dealing with distributed memory machines add new dimensions to this problem. We will describe our preliminary efforts in this area.

2 State-Space search

In a state-space search problem, one is given a starting state, a set of operators that can transform one state to another, and a desired state. The task is to find a sequence of operations that transform

*This research has been supported in part by the National Science Foundation under grant number CCR-89-02496.
†Address from Sep. 1990: Dept. of Computer Science, Oregon State University, Corvallis, OR-97330

facilities. With this, were able to obtain very good speedups for many depth first search problems. Other work on dynamic load balancing for this problem includes that of Kumar and Rao [12, 8].

When one is interested in any one solution, this parallelization technique leads to problems. If we search two successors of a state (assume there are only two for simplicity), the solution may lie in the sub-tree of either node. If it lies in the sub-tree of the first node, the work under in the second sub-tree will be wasted. Exploring the two subtrees in parallel is thus speculative - we may not need both those sub-computations. This fact, and the resultant speedup anomalies were noted in a branch-and-bound search which is closely related to depth-first search, by Lai and Sahani [9].

One may get deceleration anomalies where adding a processor may slow down the search process in finding a solution. This may happen because the added processor may create some "red herring" work that other processors wasted there time on. In extreme cases, this may lead to detrimental anomalies, where p processors perform slower than 1 processor doing the search. It is also possible to get acceleration anomalies: a speedup of more than p with p processors. This can happen because the added processor picked a part of search tree that happened to contain the solution. Kumar et al noted this in the context of parallel depth-first search. they reported a speedup varying between 2.9 to 16 with 9 processors for a 15-puzzle problem [12].

We started with the dual objectives of (1) ensuring that speedups are consistent - i.e. do not vary from run to run and (2) ensuring that the speedups increase monotonically with the number of processors, preferably being as close to the number of processors as possible. With that objective, it is clear that all the work that is done by the sequential program is "mandatory" whereas all the other nodes not explored by the sequential algorithm are "wastage".

Our scheme, described in [15] is based on bit-vector priorities. Each node in the search tree is assigned a priority. Priority bit-vectors can be of arbitrary length, and their ordering is lexicographic - the lexicographically smaller bit-vector indicates higher priority. The priority of the root is a bit-vector of length 0. If the priority of a node is X, and it has k children, then the priority of the i'th child is X appended by the $\lceil logk \rceil$-bit binary representation of i. (Thus, if a node with priority 01101 has three children, their priorities will be 0110100, 0110101, and 0110110, from left to right.) It can be shown that lexicographic ordering of these priorities corresponds to left-to-right ordering of the nodes in the tree. To be sure, there is a loss of information in the bit-vector representation: A node with priority 0110110 may be at level 7 of a binary tree, or level 3, with the top-level branching factor of 2, and the next two (grand-parent and parent of this node) with a branching factors of 7 and 5 respectively, among many other possibilities. Fortunately, this loss of information does not destroy the left-to-right ordering in a specific tree, and saves much in storage and comparison costs over a scheme that assigns a fixed number of bits to each level.

factor, the duplication cost is at most 100%, which is tolerable considering the significant memory savings.

As each iteration of IDA* is a depth-first search, it can be parallelized using the techniques described above. Kumar et al in [12]. were the first to demonstrate parallel schemes for this problem. Their results did exhibit speedup anomalies, and they reported speedups to all solutions (as their primary interest was to demonstrate the efficacy of their load balancing scheme). Note that there may be multiple optimal solutions in the last level. Even when there is one, it may be found before the whole tree up to that level is explored. Thus the notion of speculativeness prevails in this context too.

Our prioritization techniques described above were successful at getting consistent and monotonic speedups for this problem. However, the speedup with these techniques alone are not as high as they could be, (although for each iteration, we obtained close to the best possible speedups). The difficulty is that the parallelism in this problem increases and decreases in waves with each iteration. At the beginning of each iteration the parallelism is low. It increases quickly to occupy all the processors, and then trails off toward the end of the iteration. A substantial improvement was obtained by allowing multiple iterations to run together. This has to be done with some care, as work in iterations beyond the iteration containing the optimal solutions contributes further to wasted work. This was handled by assigning a non-empty increasing bit-vector priorities to the root nodes of successive iterations using an interesting encoding scheme. (See [4] for details.).

This encoding scheme solves the problem of assigning increasing priorities to successive roots, without knowing an upper-bound on how many iterations there will be. The encoding must also ensure that each node in one iteration receives higher priority than all nodes in the next iteration. With that, we were able to "soak up" the computing resources during the previously idle periods without increasing the wastage, and produce almost perfect speedups even for small-size problems.

3 Logic Programming

A Pure Logic Program is a collection of predicate definitions. Each predicate is defined by possibly multiple clauses. Each clause is of the form: $H :- L_1, L_2 ... L_n$, where the L_i's are called the body literals. (A literal is a predicate symbol, followed by a parenthesized list of terms, where a term may either be a constant or a variable, or a function symbol followed by a parenthesized list of terms). A clause with no body literals is called a fact.

A computation begins with a query, which is a sequence of literals. A particular literal can be solved by using any of the available clauses whose heads unify with the goal literal. In the problem-solving interpretation of a Logic Programs, each literal corresponds to a (sub) problem,

literals. For example, consider a clause with four literals, with the dependence graph represented by:

$$h(I,T) \ :- \ p(I,X) \ \rightarrow \ (q(X,Y) \ // \ r(X,Z)) \ \rightarrow \ s(Y,Z,T).$$

when an instance of this clause is activated, an initial binding tuple with I bound to some value, and other variables unbound, is created. One OR process for solving p with this initial binding of I is then fired. For every value of X returned by p, one q and one r process is fired immediately. Thus there may be multiple q (and r) processes active at one time. Each value of Y returned by q is combined with compatible values of Z (i.e. those that share the same X value) returned by the corresponding r process, and for each consistent combination, a s process instance is fired. Each OR process, given an instantiated literal, simply fires off REDUCE processes for each clause that unifies with the literal, and instructs them to send response to its parent REDUCE process. Thus, the process tree looks similar to a proof tree, rather than to an OR tree (or SLD tree). This fact is important in understanding (as well as designing) the scheme we proposed.

In the compiled implementation of ROPM, the requests for firing processes were stored and serviced in LIFO fashion. On (small) shared memory system, this was done using a central shared stack, whereas on distributed memory machines, a separate stack was used on each processor, and a dynamic load balancing scheme moved such requests from one processor's stack to another's. Although this strategy resulted in good use of memory space, it had one drawback (if one is interested in just one solution): all the solutions tended to appear in a burst toward the end of the computation, for problems that involve AND as well as OR parallelism. It is easy to see why, with an example. Suppose there is a clause with two AND-parallel (i.e. independent) literals, p and q. When the clause fires, it pushes p and q process-creation requests on the stack. Assume p is on top, without loss of generality. Literal p may have a large sub-tree, with many solutions, and so all the processors in the system may be busy working on p. This will result in production of all solutions to p before any solutions to q. (Of course, toward the end a few processors will be working on q while others are finishing up p). However, from the point of view of reporting first solution faster, the system should focus attention on q as soon as one solution from p is obtained. In addition, if there are two alternative computation-intensive clause for p, we should have the system concentrate its resources on one clause (and its subtree) rather than dividing them arbitrarily among the two.

The solution we proposed used bit-vector priorities, with the root having a null-priority. An OR process with priority X assigned a priority to each of its child, by appending the child's rank to X (as described in the section on state-space search). A REDUCE process used a more complex method for assigning priorities. The AND parallel literals, such as p and q in the example above, received identical priorities. The literals closer to the end of the dependence graph received higher

only for OR parallel programs. A synthesis of these is needed. We are currently working on this problem. We already have a simple scheme [14] that is sufficient to ensure consistent and linear speedups for many AND/OR problems. Other schemes, that involve dynamic changing of priorities are being developed and implemented.

4 Distributed Memory Machines

As mentioned above, priorities decouple scheduling from the tree representation itself. On distributed memory machines, this property is useful. We can now concentrate on how to make sure that high-priority nodes (are more likely to) get processed before the low priority ones. A scalable strategy involving multiple priority queues (instead of a single centralised one) has been designed to work for large shared shared memory machines, and is being implemented. A series of strategies are being designed, analyzed, implemented and tested for the distributed memory machines. The preliminary results are promising. A certain degradation, compared to small shared memory machines, is to be expected because without a central queue, processors are more likely to miss the highest priority nodes. However, we believe that such effects can be contained to a small constant degradation, allowing us to retain the important properties of scalability, monotonic speedups and (relative) consistency.

5 Discussion

We have outlined some of the ongoing work already. In addition, we plan to work on many other domains where speculative work is present. These include theorem proving, branch-and-bound computations, game-tree search etc. Each of these domains presents a different sets of constraints and challenges. For example, a left-to-right strategy will be harmful for game-tree search, because it will often carry out work that is pruned in sequential alpha-beta search. Unlike the state-space search, the wasted work is not separated by a left-to-right boundary.

The implementations of all the schemes were done in a machine independent parallel programming system called Chare Kernel. It supports dynamic creation of medium grained processes that communicate via messages and via the "specific information sharing primitives" provided in the Chare Kernel language. The kernel also supports bit-vector priorities.

The strategies described in this paper have been embodied in software that is available for researchers to use. ROLOG, the parallel Prolog compiler, SearchPack, a package for State-Space search for first solution, and the Chare Kernel parallel programming system are all available from the authors.

Killing Processes in Qlisp

Ron Goldman
Lucid, Inc.

Abstract

Qlisp, a dialect of Common Lisp, is a multiprocessing programming language which is suitable for studying the styles of parallel programming at the medium-grain level. After a brief description of the Qlisp language this paper discusses various issues involved in terminating processes that are no longer useful. Various methods to identify and kill processes are described.

1. The Qlisp Language

Qlisp, a dialect of Common Lisp, is a multiprocessing programming language which is suitable for studying the styles of parallel programming at the medium-grain level. Qlisp was initially designed in 1984 by John McCarthy and Richard Gabriel [2] while they were affiliated with the Lawrence Livermore National Laboratory's S1 Project. The S1 was to have been a 16-processor multiprocessor, with each uniprocessor being a Cray-class supercomputer. Until 1987 the only implementations of Qlisp were interpreter-based simulators. Since late 1987 we have been engaged in implementing Qlisp [3], first on an Alliant FX/8 parallel computer and now on an Encore Multimax. Our implementation is based on Lucid Common Lisp, a commercial Common Lisp system.

Qlisp uses queue-based multiprocessing. The programmer must indicate explicitly when parallelism is possible by inserting special parallel constructs in the program. When a running program executes a statement specifying parallelism, it adds a collection of new tasks to a queue for subsequent evaluation. When a processor completes a task it goes to this queue for its next task. Basing parallelism on runtime queues means that a program is not written or compiled for a specific number of processors. The number available could even change during the course of a computation. Tasks need not be of similar length, since a processor finishing a short task merely takes another from the queue.

1.1 Futures

Whenever a new process is created to perform some computation, the process will have associated with it a special datatype called a future [1]. This future is a promise to eventually deliver the value that is being computed by the process. Initially the future has no value and is unrealized. The future is realized when the process associated with it finishes its computation. If some other process needs to know the value of an unrealized future in order to perform some operation (such as addition), then it must block and wait until the future has been realized. However many operations, such as cons, assignment, or parameter passing, only require a pointer to the future and do not need to wait for it to be realized.

This research was supported by DARPA under contracts N00039-84-C-0211 and DACA76-89-C-0003.

1.3 QLET

The primary means of introducing parallelism into a Qlisp program is the qlet construct, which is used to evaluate a number of arguments to a let-form in parallel. Its form is:

(qlet (:parallelp *prop*) ((x_1 arg_1)...(x_n arg_n)) . *body*)

The form *prop* is again a propositional parameter that is evaluated first. If its value is nil, then the qlet behaves like an ordinary let in Common Lisp: The arguments $arg_1 \ldots arg_n$ are evaluated, their values bound to $x_1 \ldots x_n$, and the statements in *body* are evaluated.

If *prop* evaluates to any non-nil value, then the qlet will spawn a number of new processes, one for each arg_i, and add them to the queue of processes waiting to run. If the value of *prop* is not the special keyword :eager then the process evaluating the qlet will wait until all of its newly created child processes have finished. When the values for $arg_1 \ldots arg_n$ are available, the parent process will be awakened, the values bound to $x_1 \ldots x_n$, and the statements in *body* evaluated.

The following is an example of one way to write parallel factorial using qlet:

```
(defun pfact (n depth)
  (labels ((prod (m n depth)
             (if (= m n)
                 m
                 (let ((h (floor (+ m n) 2)))
                   (qlet (:parallelp (> depth 0))
                         ((x (prod m h (- depth 1)))
                          (y (prod (+ h 1) n (- depth 1))))
                         (* x y))))))
    (prod 1 n depth)))
```

Note that labels is a Common Lisp construct used to define locally named, mutually recursive functions, in this case one called prod. This internal function prod computes the product of integers from m to n inclusive. It does this by dividing the interval m–n into two approximately equal parts, recursively computing the products of the integers in those two intervals, and then multiplying the two results.

The cutoff depth is used to control the number of processes created. Because two are created for every recursive call in prod, at most $2^{depth+1} - 2$ processes will be spawned. Notice that the propositional parameter to qlet simply looks at the value of depth.

In the case that *prop* evaluates to the special keyword :eager then the process evaluating the qlet will not wait for the processes it has just spawned to complete the evaluation of the arguments $arg_1 \ldots arg_n$. Instead, it will bind each qlet variable, $x_1 \ldots x_n$, to a future and then proceed to evaluate the forms in *body*. If in evaluating *body* the value of one of the qlet variables x_i is required, the process evaluating the qlet will wait for the spawned process computing arg_i to finish. If the value has already been computed, no waiting is necessary.

1.5 Process Synchronization

Qlisp also provides a number of lower-level constructs such as locks, events, and semaphores to allow the programmer to specify explicit synchronization among processes. As we continue to improve Qlisp it is hoped that new higher-level Qlisp constructs will obviate the need for programmers to use the low-level locking functions.

2. Killing Processes

Because Lisp programs (and symbolic computation in general) are highly recursive, they can very easily generate a large number of parallel tasks—the opportunities leap out. Because any real multiprocessor will have only a finite number of processors, and because the cost of creating and maintaining a new process is non-zero, all Qlisp constructs have a parameter that can be used during runtime to limit the degree of multiprocessing. We need only enough parallelism to keep all the available processors busy.

It is not enough to limit parallelism only at the time that processes are created. It is also necessary to get rid of processes when they are no longer useful. Each process we create consumes system resources and for efficiency we would like to eliminate a process as soon as it is no longer contributing to the overall computation. For example, when searching a large tree, once one process succeeds any remaining processes can be terminated. However, determining when a process is superfluous is non-trivial.

The traditional way that Lisp reclaims resources that are no longer being used is via garbage collection. When it can be determined that no pointers exist to an object in memory, then that memory can be reclaimed for later use. Similarly when no pointers exist to a future, then the value of that future is no longer accessible and there is no point in continuing to work on computing it, so any processes associated with the future can be killed[2]. A process closure can likewise be killed when there are no longer any pointers to it, provided that it has completed all of the previous calls to it. There are several problems with relying on garbage collection to kill no longer needed processes. First, garbage collection does not occur frequently (one hopes), so the interval can be quite long between when a future is no longer pointed to and when the process computing it is actually killed so that it is no longer using system resources. Second, if a task was spawned for effect no pointer to the associated future may ever be retained, as the value will never be used. If a computation is known to be done for its side effects, then the keyword **:for-effect** can be added to the parallel specifier list of the Qlisp form that created the future so that the future is marked as not being subject to garbage collection. For example

```
(spawn (:parallelp t :for-effect t) (call-for-side-effect))
```

There is also the function **execute-for-effect** which takes a process or future as its argument and marks it so it will not be garbage collected. This way it is not necessary to

[2] When a process is killed, before it goes away, it first runs any cleanup forms established by the Lisp construct **unwind-protect**. While executing these cleanup forms the process is not allowed to create any new processes using **spawn** or **qlet**; it is as if the **:parallelp** argument is always forced to be **nil**.

qlambda process closure does a **throw**, the catch frames of the process that called the **qlambda** are searched rather than those in the process that created the **qlambda**.

When a **throw** affects several processes, they are all killed simultaneously. They all begin executing any cleanup forms and then go away; parent processes do not wait for their children to finish dying.

Exception handling can also cause processes to be killed and is treated similarly to **throw**.

Another way we are investigating to identify when processes can be safely killed, is to have the programmer indicate the processes involved in a computation, so that when it is finished or no longer needed, all the processes associated with it can be terminated. We are considering two approaches to this: heavyweight futures and process groups.

2.1 Heavyweight Futures

The construct **qlet** is an example of AND-parallelism—where there is a set of tasks to do and all of them must be completed. We also need a way to specify OR-parallelism—where there is again a set of tasks, but now when the first task is successfully completed, the other tasks can be abandoned. The initial design of Qlisp proposed to do OR-parallelism by combining **qlet** with the explicit killing of processes. This will work, but the resulting code often seems unnecessarily awkward and unclear. We now feel that providing Qlisp constructs to directly express AND/OR-parallelism will result in higher quality Qlisp programs that will be easier to write and will more clearly communicate the programmer's intent.

To do this we generalize the notion of a future to allow several processes to be associated with it, along with a combining function. As each process finishes, it calls the combining function with the value of the form it has just finished computing. When all of the processes have completed, the future will be realized. For example if the combining function is **+**, then the sum of all the values computed by the associated processes will be the value of the future; if it is **max**, then the maximum value returned by the processes will be the future's value. OR-parallelism is accomplished by also associating an end test predicate with the future: When the value computed by a process satisfies this end test, then the future will be realized immediately, and any processes associated with the future that have not yet finished will be killed. We distinguish between a simple *lightweight* future whose value is computed by one process, and the more complex *heavyweight* future where several processes are involved in computing the value of the future.

Heavyweight futures are created by using an extended definition of **spawn**, which accepts arguments to specify a combining function, an end test, and multiple forms to be evaluated. Additional processes can be added to a heavyweight future by passing the same future to several calls to **spawn**. The following illustrates this. The problem is to find the minimum for a function of one real variable within a given interval. The strategy is to break up the interval into n equal subintervals and to have each process search its subinterval for a local minimum. The mesh should not be any finer than the value supplied by the parameter `delta`.

An AND-Parallel Distributed Prolog Implementation

Ian Olthof*

July 23, 1990

1 Introduction

Few logic programming languages designed to exploit AND-parallelism support both full parallelism and nondeterminism. Most languages either limit the amount of parallelism (specifically, independent AND-parallel models like RAP [Degr 84]) or restrict nondeterminism to the "don't-care" variety (concurrent logic programming languages like Concurrent Prolog [Shap 83], GHC [Ueda 85], and PARLOG [Clar 86]).

The problem that these and other AND-parallel languages face is that of handling parallel backtracking, specifically among goals that share variables. Both approaches "solve" the problem by preventing its occurrence: the concurrent or committed-choice languages by disallowing backtracking of a shared binding; the others by disallowing the parallel execution of goals sharing a variable, thus simplifying backtracking.

Backtracking in the presence of shared variables can be done, and at a reasonable cost. The key to accomplishing this is to impose some (total) ordering on the goals in the search tree and to establish binding precedence based on that ordering. (In fact, sequential Prolog imposes just such an ordering temporally – specifically, a depth-first ordering.) A goal with lower binding precedence will be backtracked before one with higher precedence. Only when a lower-precedence goal can find no more solutions does it require a higher-precedence goal to backtrack.

*Department of Computer Science, The University of Calgary, 2500 University Drive N.W., Calgary, Alberta T2N 1N4, Canada. *ph:* (403) 220-7194, (403) 245-0676; *email:* olthof@cpsc.ucalgary.ca

- to provide a base upon which a number of optimizations can be built, tested, and evaluated.

As it exploits stream AND-parallelism, the interpreter requires some form of interprocess communication facility to pass information about variable bindings between processes. For generality, the system was designed to use message passing so it could run with little modification on a loosely-coupled multicomputer or a shared-memory machine, as well as the system of networked Sun-3s on which it currently runs. JIPC [JADE 85] was chosen as the message-passing subsystem for its stability, reliability, and ease of use.

The use of JIPC does, however, constrain the Prolog system. Since JIPC runs under UNIX, JIPC processes are effectively UNIX processes; thus, the system was designed under the assumption that processes are heavyweight entities. Rather than having every goal become a process, only particular goals do so (specifically, those designated as parallel in the text of the input Prolog program). Because of this, the parallelism available is lessened, but the overhead of process creation and scheduling is drastically reduced. Goals that are (largely or wholly) mutually independent, as well as goals in a producer/consumer relationship, can be run in parallel with only minor annotations: the clause

```
p(X) :- q(X)@p1, r(X).
```

indicates that when the head `p(X)` is unified, `q(X)` should be run as a separate process; `r(X)` is executed as part of the original process.

The system can also take advantage of another form of annotation to prevent wasted execution: delays. These delay annotations take a form similar to NU-Prolog's *when* declarations [Thom 88], although they are currently only implemented as triggers (ie. no structures are allowed in the declarations, only variables). For example, the declaration

```
?- p(A,B,C) when A and B or C.
```

allows a goal `p/3` to be unified only if its arguments are sufficiently instantiated: either the first two arguments or the last argument must be bound to a nonvariable.

Beyond these annotations, parallelism is completely transparent to the user; further, no preprocessing is necessary before running a program.

- it must cancel outgoing messages whose timestamp is greater than the given time;
- it must restore its internal state to what it was at a time just before the given time.

Clearly, then, some form of state-saving is necessary. A Time Warp process uses three queues to do this: an input queue (IQ), and output queue (OQ), and a state queue (SQ). The IQ contains (in timestamp order) incoming messages for the process. The OQ holds *negative* copies of all messages sent out by the process; a message is cancelled simply by sending out its corresponding *anti-message*. The SQ contains "snapshots" of the process at various virtual times; the internal state can be reconstructed using these snapshots.

Receipt of an anti-message may also cause a rollback. If its corresponding positive message is on the IQ but not yet received, the two messages can just "annihilate" each other; if the positive message has been received, the system must perform a rollback to the time of that message before annihilation may occur.

Given this description, it is clear that the mechanics of Time Warp are well suited to a Prolog system. When a variable is bound, a positive "BIND" message is sent; when backtracking a binding, a negative "ANTI-BIND" message is sent to annihilate the BIND. The state queue can be implemented quite nicely as part of the backtrack trail; so too can the output queue, since output messages are coincident with variable bindings. The input queue must be kept separate, since the trail can contain only "received" messages.

The only addition that is needed is a FAIL message, so a lower-precedence process that has exhausted its search tree can cause a higher-precedence process to backtrack.

4 The Message-Passing Subsystem

One drawback to using the Jipc subsystem is that it is a synchronous protocol in which a process sending a message is blocked until the receiver replies. The Prolog system requires an asynchronous protocol, allowing a process to resume execution immediately after sending a message. Thus, some form of mediation is required between the two systems.

In this implementation, such mediation is handled by ear processes – Jipc processes whose purpose is to make communication between interpreter

Because of the delay conditions for =/2, dereferencing is greatly reduced:

```
A → 5; B → 5; C → 5.
```

The optimization comes as a side benefit: long dereference chains are no longer necessary – a variable can be dereferenced in no more than three steps.

The second optimization involves trying to avoid rollback on receipt of an out-of-order BIND message. Instead, the system should first check whether or not the binding is compatible (noncontradictory) with the other bindings to date. If the binding is indeed compatible, there is no need to perform a rollback. Even if there is a conflict, it may not be necessary to roll back all the way to the time of the out-of-order BIND; some other BIND with a later timestamp may be at fault. This optimization should prove to be of major benefit to the performance of the system: current testing has shown that many "wasted" rollbacks are occurring.

The third optimization attempts a similar tactic to handle ANTI-BINDs: when a binding is retracted, it may be advantageous to retain the current execution path, particularly if it would just be recreated. This is more complex than the BIND optimization, however, since previously-rejected paths will have to be retried. (When a new binding is added, a path that failed previously will still fail; this is not necessarily the case when a binding is removed.)

6 Testing and Evaluation

Currently, a version of the system that implements the dereference optimization (but not the others) is being tested. When it becomes more stable and the statistics start looking meaningful, the BIND optimization will be put in and that version tested. The ANTI-BIND optimization, being more speculative and more complex, will likely not go beyond the algorithm design stage.

Statistics from the system will be compared with those from a sequential version of the system (or from the parallel system running on input programs that are free of parallelization annotations). Statistics gathered will include the following:

- running time – no earth-shattering execution times are expected here, since the code is interpreted and communications are fairly slow. Still,

References

[Clar 86] Clark, K.L., and Gregory, S., "PARLOG: Parallel Programming in Logic," *ACM TOPLAS*, 1(8), 1986, pp. 1–49.

[Clea 87] Cleary, J.G., Unger, B.W., and Li, X., "A Distributed AND-Parallel Backtracking Algorithm Using Virtual Time," Research Report 87/281/29, Department of Computer Science, University of Calgary, October 1987.

[Degr 84] Degroot, Doug, "Restricted AND-Parallelism," *Proceedings of the International Conference on Fifth Generation Computer Systems*, 1984, pp. 471–478.

[JADE 85] JADE Project, *JADE User's Manual*, SRDG, University of Calgary, 1985.

[Jeff 85] Jefferson, David R., University of Southern California, "Virtual Time," *ACM TOPLAS*, 7(3), July 1985, pp. 404–425.

[Shap 83] Shapiro, Ehud, "A Subset of Concurrent Prolog and Its Interpreter," ICOT Technical Report TR-003, February, 1983.

[Somo 88] Somogyi, Z., Ramamohanarao, K., and Vaghani, J., "A Backtracking Algorithm for the Stream AND-Parallel Execution of Logic Programs," Technical Report 87/10, Department of Computer Science, University of Melbourne, May 1988.

[Tebr 87] Tebra, Hans, "Optimistic AND-Parallelism in Prolog," *Parallel Architectures and Languages Europe*, in *Lecture Notes in Computer Science*, 2(258) Ed. de Bakker, J.W., Nijmen, A.J., Treleaven, P.C., 1987, pp. 420–431.

[Thom 88] Ed. Thom, J.A., and Zobel, J., *NU-Prolog Reference Manual, Version 1.3*, Technical Report 86/10, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

[Ueda 85] Ueda, K., "Guarded Horn Clauses," ICOT Technical Report TR-103, June 1985.

Task Switching in OM

John Conery
University of Oregon

A working paper to be presented at
The Workshop on Parallel Implementation of
Languages for Symbolic Computations
July 30–31, 1990, Eugene, Oregon

Abstract

OM is a virtual machine for OPAL, a simple logic programming language based on the AND/OR Process Model. This working paper describes the methods to organize processes and messages, and outlines two approaches to reducing the process management overhead: efficient implementation of task switching, and optimized communication patterns in the tree of processes. The paper also describes how the scheme will be implemented on nonshared-memory multiprocessors.

1 Introduction

In 1987 Dave Meyer and I designed an abstract machine for parallel programs of the AND/OR Process Model. The machine is named OM, for OPAL Machine.[1] Our goal was to design a machine that would support our parallel model much the same way the WAM supports Prolog, mainly by making unification and control more efficient by doing much of the work at compile time. The next year we had a version that was running OR-parallel programs, but was very inefficient in several areas. A few months ago we finished a new implementation which includes AND-parallel control and addresses most of the problems in the earlier implementations.

Process management remains a critical issue in the latest machine. The AND/OR Process Model is an Actors style model, in which independent

[1] OPAL stands for Oregon Parallel Logic.

The basic cycle in the execution of a logic program is for a processor to get the next message from its local message queue, install the process that will receive the message as the current process, and then execute a sequence of OM instructions that implement the process transformation. Some of these instructions will create messages for other processes. These instructions trap to the kernel, which determines whether the recipient is local or not. Messages to local processes are added to the end of the local queue, but nonlocal messages are forwarded to the kernel on the node where the process resides.

The state of an OM processor is defined by sixteen registers. Several, such as the program counter and instruction register, are self explanatory. The `E`, `A`, `R`, and `S` registers point to stack frames, which are contiguous vectors of terms. These registers are used in unification in the head of a clause. The two registers which are of most concern to this paper are the `P` and `M` registers, which hold pointers to the current process and the message that triggered its execution. We use a notation in which either symbolic names or integers are used to index portions of a structure pointed to from a machine register. For example, `P[self]` refers to the process ID of the currently executing process, and `P[desc[2]]` is the ID of the second descendant of the current process.[2]

One of the fields of each process is the address of the virtual machine code that implements process transformations. At the head of each process is a vector of "port" instructions, which are always arranged in the same order and indexed by message type. A process handles a message via a branch to the port instruction for that type of message.

A task switch consists of pulling a message from the front of the message queue and putting a pointer to it in the `M` register. Then the state vector of the process that will handle the message, which is identified by `M[to]`, the "to" field of the message, is put into the `P` register. The last step is to set the program counter `PC` to the address of the port instruction, which is the address of the code for this process indexed by the type of this message, *i.e.* `P[code] + M[kind]`. The port instruction will restore more machine registers as they are required. For example, when an AND process receives a success message, the `and_success_port` instruction restores the current process environment by putting a pointer to the environment into the `E`

[2]The notation used in the implementation depends on the programming language being used. For example, in the current C implementations, the macros `Self` and `Desc(i)` expand to `p->p.self` and `p->and.desc[i].id`, respectively

3 Implementation of Task Switching

The newest implementation of task switching is a straightforward implementation of the method outlined above. Processes are arranged in an AND/OR tree, where a pointer to a node is a four-tuple <PE,state,address,key>. The PE field is the unique ID of the processor that owns the node, which is not necessarily the processor that created it. This field is used when routing inter-PE messages. The state and key fields can be ignored for now. The address field is the local heap address of the state vector of the process; it is this address that gets loaded into the P register of the machine.

Each message has at least two of these process ID structures, identifying the sender and recipient of a message. Other process IDs, used for continuations, will be described below. Since a message contains a direct pointer to the process, and the process contains the address of the code to that will process the message, the first level (process-independent) task switching is straightforward:

```
M  := next_message;          /* trap to OS routine */
P  := M[to[addr]];           /* address of recipient */
PC := P[code] + M[kind];     /* address of port instruction */
```

The next_message procedure is defined at the OS level, and simply unlinks the next message from the front of a queue. If there are no messages, it traps to a kernel routine which waits until work (in the form of a message for an existing local task or a new task) arrives from another node. In those cases where there is a message in the local queue, the task switch is implemented by four assignments.

As mentioned earlier, the rest of the task switch after setting the PC is process-dependent, and done in the port instruction that handles the message. Some processes are very simple, and no further action is required to restore the state, while others, like the start port of a process, require the restoration of several environment register pointers.

In a nonshared memory multiprocessor implementation, the tree will be spread across the local memories of the nodes in the machine. A dynamic task allocation scheme we used in earlier implementations will be used again in the new system. When a new process is created, it has very little state information. We call these new states "seeds." The start_port instruction of a process expands a seed and initializes the state vector with information specific to that instance, for example the number of potential descendants.

clause is a nonunit clause, the response frame is mailed to the new AND process for the body, along with the initial environment for the process.

When a procedure call is in the middle of a clause, the argument frame and the response frame are the same frame, and the process that will receive the response frame is the parent AND process that is making the call. In other words, the parent AND process is the success continuation, since it is the process that will receive the success message from below when the goal is solved. However, when the procedure that is being called is the last goal in the body, the response frame is the frame that the AND process was given by its own parent. In other words, when the new goal is solved, there is no reason to send a success message back to the AND process, since it will just turn around and send a success back up the tree. By sending a different success continuation to the new OR process, it will mail its results directly to an ancestor goal, and cut out several unnecessary messages and task switches.

For calls to goals in the middle of a clause body, the new scheme saves one message per call. Instead of four messages per call, there are now three: one to start the OR process, one to start the AND process, and a success from the AND that goes directly back to the calling process. When the goal being solved is the last in a clause, the success message can go arbitrarily far back in the process tree, and many more messages and process switches are prevented.

The definition of "last call" is interesting. In Prolog, last call means the last (rightmost) call in a clause body. However, in our AND-parallel system, there may not be a last call in a clause. There are often situations where an AND process might start two or more OR processes in parallel, and then succeed when all have been solved. In these situations, none of the OR processes can be considered a last call. The AND process has to be a control point that makes sure all goals are solved before sending a success back to its grandparent.

Our compiler generates data dependency graphs for every clause body. If there is a single goal on the last level of the graph, it can be considered the a last goal, and the OR process for this goal is started with a `start_last_or` instruction. This instruction makes a new OR process and passes it the AND process's own response frame as a success continuation.

References

- Conery, J.S. *Parallel Execution of Logic Programs.* Kluwer Academic Publishers, 1987.
- Conery, J.S. Binding environments for parallel logic programs in non-shared memory multiprocessors. *Int. J. Parallel Programming 17*(2), Apr 1988.
- Lin, F.C. and Keller, R.M. The gradient model load balancing method. *IEEE Trans. Software Engineering 13*(1), Jan 1987.
- Meyer, D.M. and Conery, J.S. Architected failure handling for AND-parallel logic programs. *Proceedings of the Seventh International Conference on Logic Programming*, (Jerusalem, 1990).

one simply by backtracking up the public part of the branch. The team will only consider working on other branches when all the work on the current branch is exhausted.

When a team runs out of or-parallel work on its branch, it asks one of the active teams to provide it with access to another branch. Each active team maintains an estimate of the number of live parallel nodes on its branch, known as its *richness*. The richest team is asked to make all of its branch shared (known as *making work public*), allowing the team to start work at the bottom of the branch. This strategy will concentrate teams in "deep" regions of the or-tree, and by using backtracking, the teams should be able to find more assignments for themselves before they need to look again to the richest team for work. In this way we can minimise both the task switching overheads and also the amount of or-parallel "housekeeping" work each team has to perform.

A goal of the Andorra-I system is to implement all standard Prolog built-in predicates, preserving their sequential semantics. To achieve this, we delay the execution of a call to a side-effect predicate until its branch becomes leftmost in the whole or-tree. We implement this delay by allowing teams to suspend branches which are not leftmost. There are some special predicates, for example those used in the implementation of bagof, where it is sufficient to ensure that the branch is leftmost within a subtree.

To preserve the sequential semantics, a cut or a commit will not go ahead if there is a chance of it being pruned by a cut with a smaller scope. In the present implementation we do not maintain information on branches being in the scope of a cut. We ensure the semantics of cut and commit by delaying their execution until they are leftmost in the proper subtree, which means that currently commit is implemented as a cut. The most recent interface provides information about the presence of pruning operators in clauses. We plan to use this information to improve the implementation of cut and to implement commit properly.

When executing a pruning operator we need to prune branches, some of which may exist as unexplored alternatives while others may already be in the process of being explored. To prune a partially explored subtree we need to know which teams are working there. For this purpose, all nodes have an ownership field, giving the identity of one team which is working at or below that node. A team wishing to prune a subtree will send a cut signal to the owner of the subtree's root node, passing on responsibility to the owner to similarly find and interrupt all other teams in the subtree. These teams will then backtrack out of the pruned subtree.

3.1 Engine Speed

Table 1 compares Andorra–I/single_worker with C-Prolog to show the basic speed of Andorra–I regardless of parallelism. The first column for Andorra–I refers to a version without support for or-parallelism, while the second refers to the full implementation. The programs used were naïve reverse of a list of 200 elements, a Boyer theorem prover [6], a muiu theorem prover that was taken from an idea by Hofstadter [8], and a Prolog parser. The results were obtained on a Sequent Symmetry.

	C-Prolog	And-parallel Andorra	Full Andorra
nreverse_200	3.45	3.98	5.81
boyer	8.67	NA	8.37
muiu	0.41	NA	1.15
Prolog_parser	1.15	NA	1.1

Table 1: RUNTIMES (IN SECONDS) FOR C-PROLOG AND ANDORRA

Looking at the times for nreverse we see that the and-parallel Andorra is comparable in speed to C-Prolog. When we add in support for or-parallelism , we impose a relatively high overhead, about 50% in average. We believe that we can reduce this overhead by further optimisation of Andorra–I. Our system also has the added overheads of using a goal stack and having to handle suspension.

3.2 Search space

One of the most interesting aspects of the Basic Andorra model is its ability to reduce the search space. As an example, in Table 2 we compare the number of inferences needed in Andorra–I versus Prolog. We used programs for colouring a map, the zebra puzzle [12], two versions of the eight queens problem, and the resource allocation program (fly_pan) described by Bahgat [2].

	Andorra–I	Prolog
map	1047	5003
zebra	5757	19404
8queen_prolog	214918	383146
8queen_pandora	8496	133612
fly_pan_1	1517	10539
fly_pan_2	7800	358696

Table 2: NUMBER OF INFERENCES MADE BY ANDORRA–I AND PROLOG

The zebra puzzle deserves a more detailed explanation. Our version is based on the CHIP version as presented in [14]. For Andorra–I, we used an explicit representation of the domain

	Number of teams				
	1	2	4	6	11
queens	1	1.9	3.7	5.4	8.9
map	1	1.7	3.0	4.1	6.1
sendmoremoney	1	1.9	3.2	4.4	6.2
5x4x3	1	2.0	3.9	5.9	10.3
zebra	1	1.9	3.3	4.3	5.2
cypher	1	1.9	3.5	4.7	7.1

Table 4: OR-PARALLEL SPEEDUPS

3.5 Combining And- and Or-parallelism

By varying both the size of teams and the number of teams we can assess the effects of exploiting both and- and or-parallelism together. We did this on two programs, fly_pan and sets. Sets is a program which computes a number of subsets from a larger set and then merges the subsets into a single set.

Tables 5 and 6 show the speedups which were observed on these two programs. The results for the combination of and- and or-parallelism were obtained by running the program with all combinations of numbers and sizes of teams, the figures presented being the best time for the specified number of processors, regardless of the organisation of teams and workers. The same results are presented graphically in Figures 1 and 2.

Type of Parallelism	Number of processors									
	1	2	3	4	5	6	7	8	9	10
⋆ Comb. and/or	1	1.64	2.13	2.43	2.90	3.10	3.55	3.55	3.76	3.88
◇ or-Par.	1	1.44	1.65	1.83	1.87	1.99	2.04	2.11	2.13	2.15
∘ and-Par.	1	1.64	2.08	2.42	2.65	2.78	2.92	2.99	3.01	3.04

Table 5: SPEEDUPS OBTAINED FROM FLY_PAN_2

Type of Parallelism	Number of processors									
	1	2	3	4	5	6	7	8	9	10
⋆ Comb. and/or	1	1.71	2.61	2.89	3.45	4.52	4.77	5.35	5.88	6.17
◇ or-par.	1	1.71	2.61	2.21	2.46	3.23	2.43	2.40	2.59	3.22
∘ and-Par.	1	1.40	2.42	2.89	3.26	3.50	3.74	3.89	4.04	4.08

Table 6: SPEEDUPS OBTAINED FROM SETS PROGRAM

We can see that in both cases the speedups when exploiting both and- and or-parallelism are greater than the speedups achievable from either kind of parallelism alone.

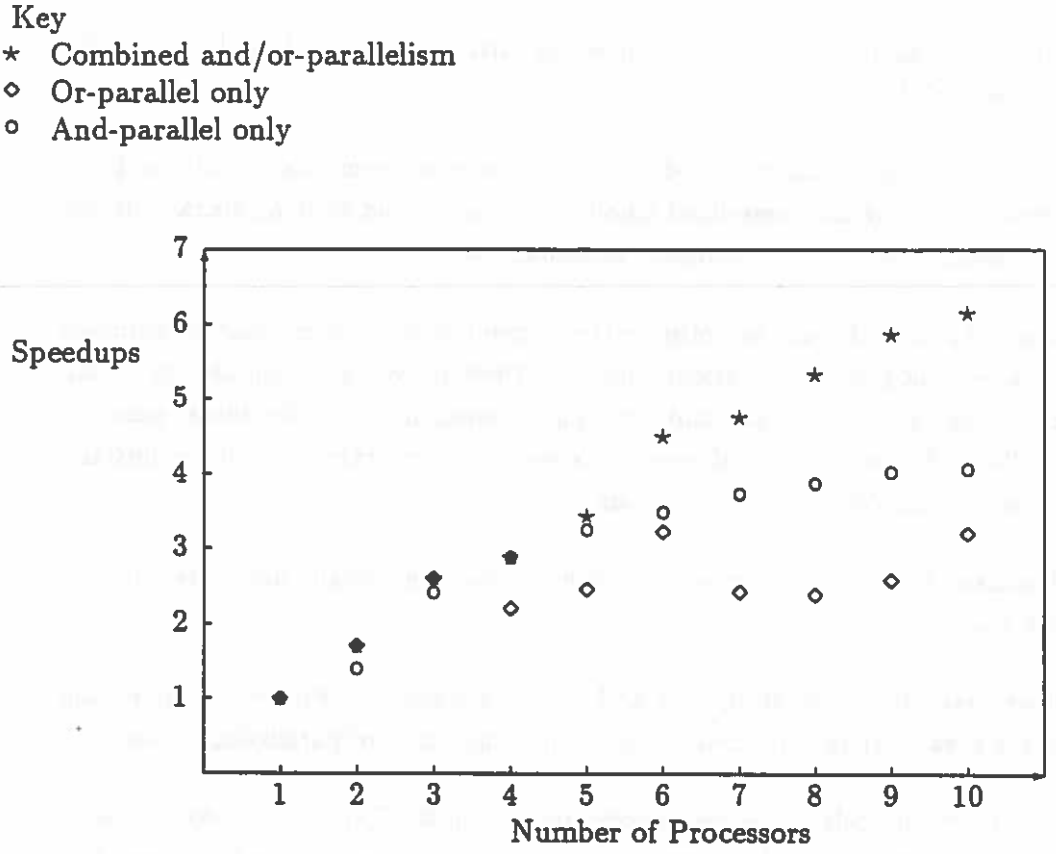


Figure 2: SPEEDUPS FROM SETS

References

[1] Khayri Ali. *Or-parallel execution of Prolog on BC-Machine.* SICS Research Report, Swedish Institute of Computer Science, 1987.

[2] R. Bahgat. *Solving Resource Allocation Problems in Pandora.* Technical Report, Imperial College, Department of Computer Science, 1990.

[3] R. Bahgat and S. Gregory. Pandora: non-deterministic parallel logic programming. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 471–486, The MIT Press, June 1989.

[4] Anthony Beaumont, S Muthu Raman, and Péter Szeredi. *Scheduling Or-Parallelism in Aurora with the Bristol Scheduler.* Technical Report TR-90-04, University of Bristol, Computer Science Department, March 1990.

[5] V.S. Costa. Compile-time analysis for Andorra-I. October 1990. PEPMA Project workshop, University of Bristol.

[6] Richard P. Gabriel. *Performance and evaluation of Lisp systems.* The MIT Press, 1985.

[7] S. Haridi and S. Janson. Kernal Andorra Prolog and its computational model. In D.H.D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 31–46, Mit Press, 1990.

[8] Douglas R. Hofstadter. *Godel, Escher, Bach: an eternal braid.* Harmondsworth: Penguin, 1980.

[9] P.J. Stuckey J. Jaffar, S.Michaylov and R.H.C. Yap. *The CLP(R) Language and System.* IBM/CMU Research Report, IBM, 1988.

[10] S. Kliger and E. Shapiro. A decision tree compilation algorithm for fcp(—,:,?). In *Proceedings of the 1988 Symposium on Logic Programming*, pages 1315–1336, 1987.

[11] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[12] Leon Sterling and Ehud Shapiro. *The Art of Prolog.* The MIT Press, 1986.

[13] P. Szeredi and M. Carlsson. *The Engine–Scheduler Interface in the Aurora Or–Parallel Prolog System.* Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[14] P. Van Hentenryck, editor. *Constraint Satisfation in Logic programming.* The MIT Press, 1989.

Performance Evaluation of a Dataflow/ von Neumann Hybrid Architecture for Parallel Execution of Logic Programs †

Surapong Auwatanamongkol

Department of Computer Science and Engineering
Southern Methodist University
Dallas, Texas 75275.

Prasenjit Biswas

Computer System Laboratory
Texas Instruments
Dallas, Texas.

Extended Abstract

1. Introduction

A stream-based data-driven execution model (LogDf) was proposed by Biswas and Tseng [BiTs88] for OR-parallel and Restricted-AND- parallel execution of Logic programs. This paper will cover a modified version of the model and a dataflow/ von-Neumann hybrid architecture [BuEk87, Iann88] to support an efficient implementation. This is not yet another architecture/execution model for parallel execution of Logic programs. Our primary objective is to evaluate the performance of the LogDf model to compare it with some of the existing parallel implementations on multiple von-Neumann processors. In order to make a fair comparison, we have provided minimal extensions to a RISC-like architecture for supporting the data-driven scheme based on variable resolution macro-actors.

It was realized quite early in the LogDf project that a dataflow model with explicit synchronization at the fine-grain instruction level [ArNi90, ItSh84] was not an efficient scheme for parallel excution of Logic programs. This decision was based on the fact that dataflow instruction sequencing for inherently sequential threads of computation did not seem to provide any performance advantage. This led to the definition of the LogDf with macro-actors to represent sequential threads of computation and explicit data-driven synchronization for enabling the execution of these macro-actors. Typical macro-actors (ie. dataflow nodes) in the earlier version represented unification, instantiation, stream-Cons etc.

Experiments with the earlier version of the LogDf model and the architecture revealed that a definition of a fixed set of macro-operators and corresponding function units (at the machine level) denied opportunities for several compile time optimizations that could normally be performed in a WAM based parallel implementation on von-Neumann processors. The current version of the multi-ring hybrid architecture is based on multiple conventional RISC-like processing elements with the additional capability to support split-phase memory transactions with I-structure stream memories. The processing elements are identical and have a fixed instruction set. The variable resolution (grain size) macro-actors representing sequential threads are compiled for these processing elements using standard register optimization techniques.

† This research was supported in part by Texas Advanced Technology Program Contract 3128 (1988) and a Research Grant from Texas Instruments (1988).

To see how a logic program can be compiled into a dataflow graph using the above mentioned actors, let us consider a set of clauses as shown below (arguments of each literal are not shown).

```
clause 1 :      p :- q, r.
clause 2 :      p :- s.

clause 3 :      q.
clause 4 :      q.

clause 5 :      r.
clause 6 :      r :- t.

clause 7 :      s.
clause 8 :      s.

clause 9 :      t.
clause 10 :     t.
```

Figure 1 shows a tree structure dataflow graph representing nested calls for an execution of a goal p. In this graph "success" and "failure" of unifications have been arbitrarily assumed to create the example. A success leaf is reached when there is no more goal literal to be solved. For each success leaf, there is a corresponding stream cell, e.g. cell a, b, c, d, e and f, which contains a pointer to solution bindings. These cells are linked together and form a stream of solutions for the goal p. If a unification required to enter a procedure for a candidate clause does not succeed, a failure leaf is reached.

In a dynamic dataflow execution model, different actvations of the same dataflow graph are distinguished by coloring of tokens flowing in the graph [ArNi90]. Different colors are assigned to identify distinct invocations. A brief description of the token coloring scheme for the LogDf is essential for understanding how the return address of a call can be passed from a Unify actor to any of its corresponding Return actors. In the LogDf the address of a descriptor is used as a color of a token in the dynamic dataflow execution [BiTs88]. This color provides an identification of a particular activation of a dataflow graph (e.g. invocation of a clause graph for a specific set of input bindings). When a token activates a Unify actor, a new descriptor is created and is used to store the color of the input token along with other relevant information (including the return address). The address of the new descriptor is used as the color of the output token. The generated color is later used in the associated Return actor to access the descriptor created by the Unify actor to retrieve the stored color for painting its output token.

3. The dataflow/ von Neumann architecture

In the original LogDf architecture [BiTs88], each node in a dataflow graph of a logic program is a macro-actor and is executed in functional units corresponding to each type of macro-actor. Since the macro-operators have fixed resolutions, the system inherits same deficiencies that plague conventional CISC architectures. Also any specialization of any dataflow node to suit a particular invocation could not be performed due to the fixed resolution of macro-operators.

To alleviate these deficiencies, we adopt a system that supports variable resolution macro-dataflow execution, which represents macro-dataflow nodes by threads of instructions. The threads are executed by conventional RISC-like processing units, and so functional units in a ring of the original LogDf is replaced with a set of identical processing units.

The new architecture provides instructions that support frequently occuring types of unifications in logic programs. Data in memories and registers are tagged to facilitate unification operations. The tags are checked and manipulated directly by some specific instructions. The data-driven execution of the original LogDf model is still supported by providing additional instructions that perform split-phase transactions on I-Structure-like memories, i.e. descriptor, S-Stream, Vframe and List/Structure memories. These split-phase transactions would provide the synchronizations between producers and consumers, and allow fast context switching during a delayed memory operation. Figure 2 shows the organization of the new dataflow/von Neumann hybrid architecture for a ring of the LogDf.

code for the macro-actor. This address is sent along inside an activating token for the actor. A frame will be allocated for the actor and the frame register will point to the beginning of the frame. When a context switching occurs during a split-phase transaction on an I-structure memory, the content of the two registers are sent along with a request token to the memory. When the split-phase transaction has been completed, a token with the continuation is sent back to reactivate the suspended execution. Upon reactivation, the two registers are restored with the values of the continuation and the execution can restart from where it left off.

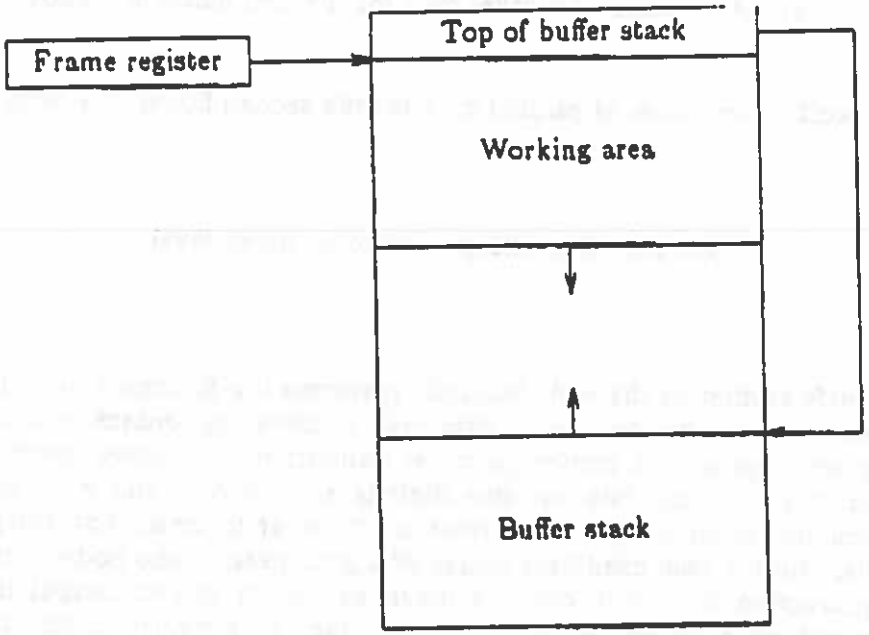


Figure 3 : Organization of a frame

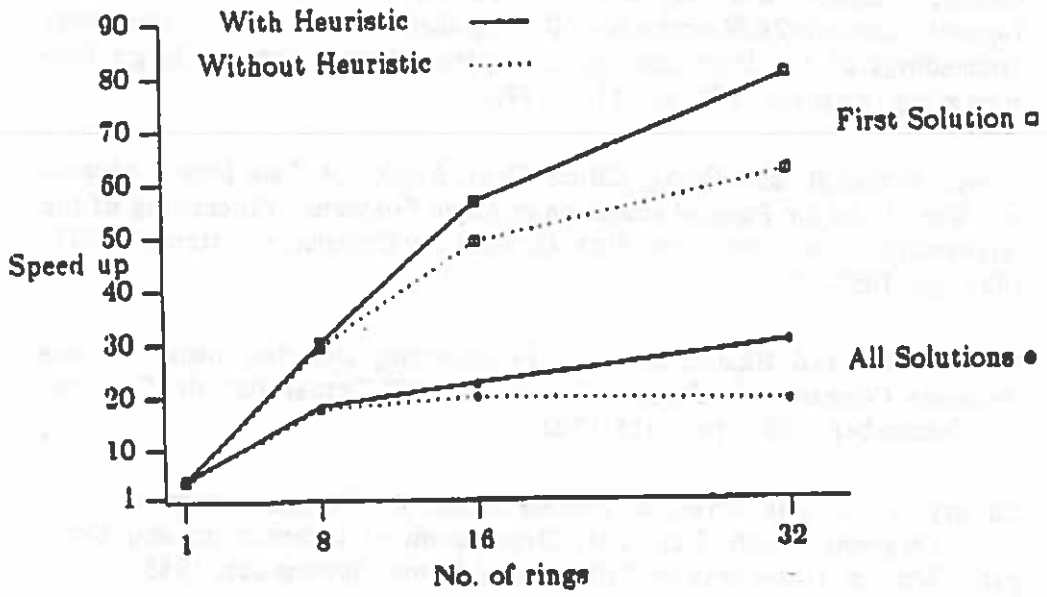
Multiple rings of the hybrid architecture are connected by a hypercube interconnection network. The communication section in each ring would provide the routing services for the interconnection network. Instructions of the dataflow/von Neumann Hybrid processor have been divided into the following categories based on types of their operations :

- S-Stream Operations
- Descriptor Operations
- Vframe Operations
- List/Structure Operations
- Register/Memory Operations
- Arithmetic/Boolean Operations
- Control Operations
- I/O Operations

scheduling strategy are not discussed in this paper. We will show speedup results from the simulation based on the following configuration :

- 10 MHz clock cycle is used
- number of processors/ring is 4,
- communication delay between two adjacent rings in the hypercube interconnection network is 2 clock cycles/word.

Speed ups on 7-Queen program (relative to performance on single processor in a ring)



Additional results will be shown at the presentation.

[HaAm84] Hasegawa, R., and Amamiya, M., *Parallel Execution of Logic Programs based on Dataflow Concept*, Proceedings of the International Conference on Fifth Generation Computer Systems, 1984, pp. 507-516.

[Hali86] Halim, Z., *A Data-Driven Machine for OR-Parallel Evaluation of Logic Programs*, New Generation Computing, Vol. 4, 1986, pp. 5-33.

[Herm87] Hermenegildo, M. V., *Relating Goal Scheduling, Precedence and Memory Management in AND-Parallel Execution of Logic Programs*, Proceedings of the Fourth International Conference on Logic Programming, 1987, pp. 556-575.

[Hogg84] Hogger, C. J., *Introduction to Logic Programming*, Academic Press, 1984.

[Iann88] Iannucci, R. A., *Toward a Dataflow/von Neumann Hybrid Architecture*, Proceedings 15th Annual International Symposium on Computer Architecture, Honululu, Hawaii, June, 1988, pp. 131-140.

[ItSh84] Ito, N., and Shimizu, H., *Dataflow Based Execution Mechanisms of Parallel and Concurrent Prolog*, New Generation Computing, No. 3, 1984, pp. 15-41.

[KeLT84] Keller, R. M., Lin, F. C. H., and Tanaka, J., *Rediflow Multiprocessing*, Digest of Papers, Spring COMPCON'84, IEEE Computer Society, 1984, pp. 410-417.

[Kowa74] Kowalski, R. A., *Predicate Logic as a Programming Language*, Proceedings of IFIP, 1974, pp. 569-574.

[NiAr89] Nikhil, Rishiyur S., and Arvind, *Can Dataflow Subsume von Neumann Computing ?*, Proceedings 16th Annual International Symposium on Computer Architecture, Jerusalem, Israel, June, 1989, pp. 262-272.

[Tsen89] Tseng, Chien-Chao, *A Data-Driven Parallel Execution Model and Architecture for Logic Programs*, Ph.D. Thesis, Department of Computer Science and Engineering, Southern Methodist University, August, 1989.

Side Effects in Prolog Programs (Extended Abstract) *

Laxmikant V. Kalé David A. Padua
David C. Sehr
Center for Supercomputing Research and Development
Department of Computer Science
University of Illinois at Urbana-Champaign
305 Talbot Lab
104 South Wright Street
Urbana, Illinois 61801

July 23, 1990

Abstract

An interesting and difficult problem facing the logic programming community is the parallel execution of common side effect predicates such as cut, asserta, assertz, and retract. Any system designed to execute existing Prolog programs in parallel needs to handle these predicates. In our previous work we have presented a method for the Or parallel execution of Prolog programs with side effects. We have written an interpreter for this system that runs on the Alliant FX/8, and have gathered some encouraging performance data.

In this paper we briefly review our method, discussing the issues of speculative parallelism and redundant computation as they arise. We conclude by presenting some results gathered from our implementation.

*The work of L. V. Kale was supported by the National Science Foundation under Grant NSF-CCR-8700988. The work of D. C. Sehr and D. A. Padua was supported in part by the National Science Foundation under Grant NSF-MIP-8410110, the Department of Energy under Grant DOE DE-FG02-85ER25001, and a donation from the IBM Corporation to the Center for Supercomputing Research and Development. This work was undertaken while D. C. Sehr held a fellowship from the Office of Naval Research.

Figure 1 shows a three clause procedure that contains a cut. In a sequential Prolog system all of the alternatives from the first clause would be tried first. If they either all failed, or successes were rejected by the user, then the clause with the cut could execute. Hence, the cut does not effect the clauses prior to the one in which it occurs. The second requirement for the cut to execute is that f(X) must have a solution. If there was more than one possible successful return from f(X), only the computation up to the first is performed. The remainder is not performed on an invocation from p(X). Moreover, if the cut executes, then the third clause for p is not tried.

In a parallel system executing the program of Figure 1 we might try all possibilities for p simultaneously, and when f is invoked, try all of its clauses as well (i.e. Or-parallel execution of p and f). If we do this, there might be some work done that would not have been in a sequential system. For example, if we work through f(X) and find several solutions, then the computation of all but the first is useless, because they will be discarded by the cut. Similarly, if the cut is executed, then any resources devoted to processing the last clause for p have been wasted. Because of this, some researchers [1, 3] have proposed techniques that do not devote processors to clauses lying below a cut. This can be restrictive when the cut does not execute. In that case, the last clause must wait until the call f(X) has failed.

More recent work [4, 5, 2] have considered the question of speculative parallelism as a scheduling issue. A speculative portion should be started if there is not non-speculative work available to be done in parallel. Thus, if there are sufficient processor resources, the third clause above might be processed in parallel with the second. The compile-time detection of speculative computations is an ongoing research problem.

3 Assert and retract

The asserta, assertz, and retract predicates in Prolog allow the run-time alteration of a program. Used incorrectly, these can be significant impediments to performance in sequential systems. However, in some cases they can improve performance dramatically. In a parallel system these predicates can similarly cause a number of dependences, seriously inhibiting parallelism. There are cases, however, where large amounts of redundant computation can be avoided by using assert and retract. There are also cases where an Or parallel system using these predicates can exploit more *consumer instance parallelism* than without them.

```
p(X) :- f(X), q(Y), consume(Y,X).

q(A) :- compute(A), asserta((q(A) :- !)).
```

Figure 3: Parallelism and redundant computation

Name	Processors 1	2	3	4	5	6	7	8
fib.pl	1.12	1.16	1.13	1.16	1.16	1.14	1.14	1.12
merge.pl	0.44	0.49	0.49	0.48	0.38	0.48	0.47	0.46
press.pl	0.53	0.59	0.62	0.63	0.63	0.61	0.61	0.62
lips.pl	0.67	1.25	1.71	2.26	2.53	2.95	3.27	3.61
mapu.pl	0.64	1.23	1.76	2.26	2.73	3.06	3.40	3.57
projgeom.pl	0.45	0.77	0.87	1.01	1.08	1.13	1.21	1.29
q8.pl	1.01	1.96	2.82	3.55	4.19	4.79	5.29	5.39
regof.pl	0.28	0.39	0.57	0.55	0.58	0.62	0.61	0.60
bagof.pl	Not available							
bagofr.pl	Not available							

Table 1: Performance compared to C-Prolog

between consume goals. If the computation of f is relatively simple and the computation of consume relatively long, this could reduce parallelism considerably.

Our method allows the processing of predicates such as q whenever they occur, and "backpatches" whenever asserts or retracts happen alter the program. Thus, for only a slight book-keeping overhead, we are able to exploit the parallelism present in this program. The speculative parallelism caused by goals being processed and subsequently retracted is controlled by the scheduler again preferring lexically earlier clauses.

4 Implementation and performance

An Or parallel Prolog interpreter that processes the standard Prolog side effect predicates has been implemented on the Alliant FX/8, and some

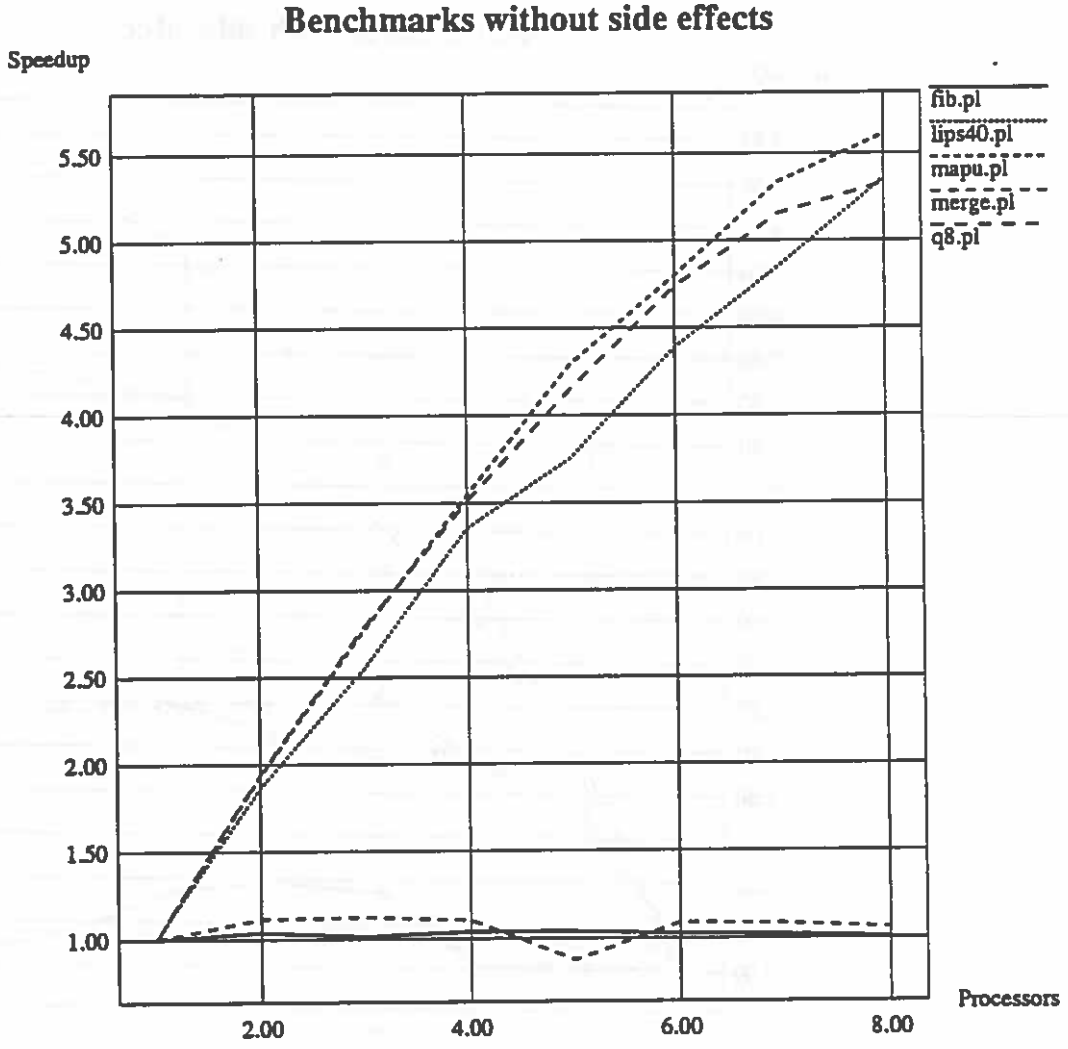


Figure 4: Pure benchmark performance

are variants of the eight queens with **bagof** implemented by **assert** and **retract**.

Monkey.pl is the monkeys and bananas problem, a search problem where a monkey tries to move a set of objects in order to reach a bunch of bananas suspended above it. Because of the heavy use of the **cut** and **if-then-else** constructs, this program exhibits little Or parallelism. The speedup is hence marginal. Press.pl is an equation solver, and it also shows little modest, in this case due to optimization to use clause indexing and many cuts. Whereas monkey.pl begins to show some signs of processors contending for locks, press.pl maintains a more level performance curve. This would appear to be because less speculative work is done because the cuts appear very early in the clauses of press.pl.

Next is the projective geometry benchmark from the workshop, projgeom.pl. In this benchmark we are trying to construct a perfect difference set. On this benchmark we are able to obtain some Or parallelism from the **member** predicate. Since the tests performed after this call are relatively simple (at least in the benchmark version), the amount of OR parallelism is somewhat limited. We do, however, see a speedup of roughly 3 on 8 processors.

Regof.pl is a solution to the latin squares problem. We are able to extract Or parallelism almost exclusively from the top-level branch that determines which method to use to construct the table. The benchmark program had several special case solutions to the problem, and the interpreter is able to gain a speedup in this portion. Since there are a limited number of choices, the speedup curve rapidly levels off to a constant speedup.

The last two problems are constructed using a *bagof* construction to save the results of the eight queens problem. Because of the semantics of C-Prolog **retract**, these could not be run under C-Prolog. The first version, bagof.pl, calls the accumulating procedure to retrieve the results of the computation. Because of our eager evaluation of dynamic predicates, more processors working on the program increases the amount of speculative work started (and hence terminated). Thus, the speedup is less than the second version, bagofr.pl, which **retracts** the accumulator clause. The version used in bagofr.pl has a speedup curve that looks very similar to the pure eight queens problem presented in the last section.