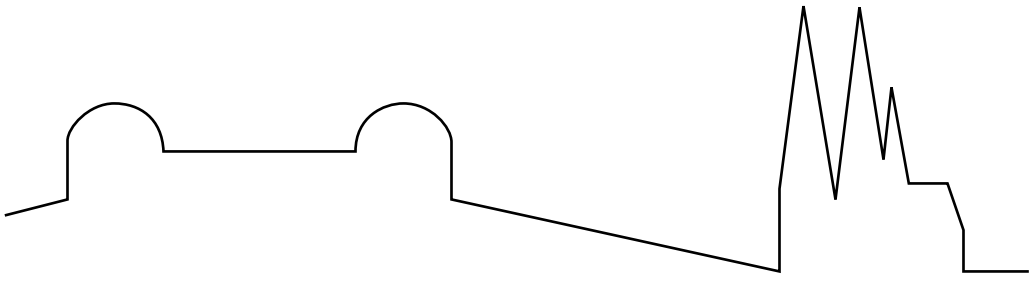
No. 16/93

A Recursion Parallel Prolog Engine

Johan Bevemyr



Thesis for the Degree of
Licentiate of Philosophy

ISSN 0283-359X

UPMAIL
Computing Science Department
Uppsala University
Box 311
S-751 05 UPPSALA
Sweden

UPPSALA THESES IN COMPUTING SCIENCE
16

A Recursion Parallel Prolog Engine

by
Johan Bevemyr

UPMAIL
Computing Science Department
Uppsala University

Thesis for the Degree of
Licentiate of Philosophy

UPPSALA 1993

Abstract

We present the design and implementation of a recursion parallel Prolog engine based on the Reform execution model. This is a data-parallel approach to parallelizing Prolog on MIMD machines. The implementation is based on a conventional Prolog engine, WAM. The engine has been extended to handle recursion parallel execution on a shared memory architecture. List and integer recursive predicates that are binding deterministic with respect to variables shared between recursion levels are parallelized. These restrictions make it possible to extend a WAM obtaining a low parallelization overhead, between 2% and 12% on the measured benchmarks.

We compare approaches to implementation decisions and discuss potential problems in our implementation.

Our parallel implementation of Prolog has been measured to execute independent AND-parallelism up to 23.15 times faster, and dependent AND-parallelism up to 22.52 times faster, on 24 processors than on one processor.

Acknowledgments

The work presented in this thesis has been discussed with Thomas Lindgren and Håkan Millroth. They have given both helpful and encouraging comments on this work. This thesis contains my contribution towards the goal of our project—designing and implementing Reform Prolog. I especially want to thank Håkan for his recommendations concerning the evaluation of the abstract machine.

Many thanks to Jonas Barklund for numerous comments that helped make this thesis more stringent.

I also wish to thank Sten-Åke Tärnlund for initiating this work by introducing me to the Reform group.

Thanks to Sverker Janson for questioning why I used a lock tag instead of an atomic exchange operation when binding shared variables.

I would also like to thank Roland Karlsson for helping me master the shared memory routines on the Sequent Symmetry.

On behalf of the Reform group I thank the Swedish Institute of Computer Science for making their 26 processor Sequent Symmetry available to us.

Finally, completing this work would have been difficult without Katrin Boberg's professional help and personal caring.

Contents

Introduction

The number of parallel computers offered for sale has increased rapidly the last few years. All major computer manufacturers are selling or are announcing new parallel computers. Most of these computers support logically shared address space; either simulated with message passing or through a physically shared memory. Often the potential parallelism is used to increase the throughput by executing distinct programs in parallel. It is rarely used to make individual programs execute faster.

There are several reasons why these computers are used almost only in this way.

1. Existing compilers are not able to utilize inherent parallelism in programs written for uniprocessors.
2. It is hard to write parallel programs when the parallelism has to be explicitly controlled by the programmer, e.g., by using *monitors*.
3. Using a single processor is sufficient for many problems.

The reason why existing compilers have problems parallelizing existing programs is that most programming languages are hard to analyze. Loops not containing procedure calls are the only constructs that have been automatically parallelized successfully in imperative languages. However, Singh and Hennessy [42] show that loop parallelization is not enough to achieve significant speed-ups in real applications.

Since the compilers cannot parallelize most real programs, a programmer is forced to write his program using operating system primitives, and in that way control the parallelism himself. This requires extensive knowledge about the system, and the resulting program is usually both complex and can only be efficiently executed on one given architecture, if at all.

One would like to have a programming language which can be executed efficiently on multiprocessor as well as uniprocessor computers, without modifying programs. It is also desirable that the programming language takes care of all underlying problems when executing a program in parallel, e.g., synchronization between parallel processes, data distribution, etc.

Prolog as a Parallel Language

One of the goals of the Fifth Generation Computer Systems project in Japan was to produce a system for parallel programming. In the preliminary report Motooka [35] defines the requirements of the future computer system. The system should, among other things, be easily programmed in order to reduce the *software crisis*. They chose the high-level language Prolog as the programming language to parallelize. The reason for this was that Prolog contains many opportunities for parallelism: AND-parallelism, OR-parallelism, and unification parallelism. AND-parallelism can be described as resolving many atoms in the resolvent in parallel. If a variable appears in two different atoms in the resolvent at runtime, then these atoms are said to be *dependent*. Resolving dependent atoms, in parallel, is referred to as *dependent* AND-parallelism. Resolving atoms that are not dependent, in parallel, is called *independent* AND-parallelism. OR-parallelism, on the other hand, corresponds to trying to resolve a given atom in the resolvent with several clauses, in parallel. In addition to containing many opportunities for parallelism, Prolog programs are easy to analyze, especially when compared with C and Fortran programs.

It turned out to be harder than expected to parallelize Prolog. A number of modifications to Prolog were proposed in order to simplify the problem. This also made it harder to write programs in these languages. The following quote is from Tick's book on parallel logic programming [46, page 410]. The book contains many programs written for different parallel logic programming systems. The quote is from the section in which he discusses how easy it was to write those programs.

> "In general, it was easy to write all of the programs in Prolog. However, it was difficult to rewrite problems without OR-parallelism ro run efficiently under Aurora. It was difficult to implement logic problems, such as Zebra and Salt & Mustard, in FGHC. This difficulty occurred because we lacked meta-logical builtins in Panda, but moreover because backtracking over logical variables must be simulated. Such difficulty is also seen when comparing the coding effort to implement complex constraints—Prolog was significantly easier than FGHC, as shown in Turtles and layered-stream programs in general."

Tick's experience is representative for most of the logic programming community. It is evident that Prolog is to be preferred. It might be possible to execute

de-evolutionized languages such as FGHC efficiently, but the programming effort involved in writing programs in these languages is in many cases much larger.

Parallelizing Repetition

There are essentially two methods for parallel programming: either different operations are executed simultaneously (*control-parallelism*, i.e, process-parallelism), or the same operation is executed simultaneously on a set of data (*data-parallelism*). Data-parallelism in, e.g., C and Fortran, is usually achieved through parallelization of repetitive operations on elements in arrays, i.e., loop-parallelization. Another common method is extending the language with special data structures on which it is possible to simultaneously apply operations to all elements in the structure, e.g., C* and *LISP.

All attempts so far to parallelize Prolog have used control-parallelism, either by mapping elements of conjuctions to different processes (AND-parallelism), or mapping elements of disjuctions to different processes (OR-parallelism). (Combinations of the two have also been used.) We propose to use data-parallelism to parallelize Prolog. Using data-parallelism simplifies scheduling and process management, among other things. Not all programs can be parallelized using this technique. However, an interesting set of programs can be parallelized. In Prolog, repetitive operations on data structures and numbers are expressed using recursion—hence this is our target for parallelization.

Recursion Parallelism

The basic idea behind recursion parallelism is to parallelize structure recursion. One might consider different variations on this theme. We have chosen to implement Reform Compilation as described by Millroth [31] (see Section 2.2). We have introduced one limitation on the predicates that we attempt to parallelize: bindings of variables shared between recursive calls of the predicate must be deterministic. This is not a severe restriction in practice.

Our implementation exploits recursion parallelism. We believe that recursion parallelism can be easily combined with other forms of parallelism, e.g., OR-parallelism. We argue that recursion parallelism is a superior way of parallelizing logic programming for the following reasons.

1. Executing recursive calls to a predicate in parallel gives a simple model for parallel execution. It is easy to understand which parts of a program are going to be executed in parallel and which are not.

2. The programmer can use the programming techniques he is used to. Recursion parallelism is optimized for executing deterministic programs—an efficient parallel program is an efficient sequential program.

3. The declarative semantics of a program is the same, regardless of whether it is executed sequentially or in parallel.

4. No modification to the Prolog language is necessary in order to implement recursion parallelism efficiently.

5. Recursion parallelism can be implemented efficiently.

Clearly these points are most relevant when comparing with committed-choice implementations. However, they do apply in varying degree when comparing with AND-parallel, OR-parallel, and AND-OR parallel implementations.

1.1 PARALLEL MACHINE MODEL

We assume a multiprocessor with *logically shared* address space in which *sequential consistency* is maintained. By logically shared we understand that processors are not required to have a physically shared memory, but might emulate the shared memory using message passing. By sequential consistency we understand that all processors observe the same interleaving of memory accesses.

We also assume the existence of an atomic exchange operation. This operation atomatically exchanges the contents of a register and a memory location, and is used for synchronization. The machine is assumed to be a MIMD (multiple instruction stream, multiple data stream) machine, this implies that different programs are allowed to execute on different processors using different data.

The MIMD machine is used in a structured way to implement data-parallelism. All processors asynchronously execute identical copies of a modified sequential WAM engine.

1.2 WARREN'S ABSTRACT MACHINE

Warren's abstract machine (WAM) [51] is an efficient machine for execution of Prolog programs. It was originally published by Warren in 1983 and has since become the most popular method for implementing Prolog. There are, by now, numerous variants of WAM; of these we have chosen the variant described by Carlsson [7, 8]. We give a brief overview of this machine in this section; for a more detailed description we refer to Aït-Kaci's book [1].

WAM is a register-oriented machine. Clauses in the database are compiled into sequences of machine instructions. These instructions specialize the head unification for each compiled clause (instead of applying the general unification algorithm). There are also instructions to encode the search rule of Prolog. When executing a predicate that has several clauses, the first clause is tried first and a *choice point* is created. A choice point is a structure containing the information necessary to

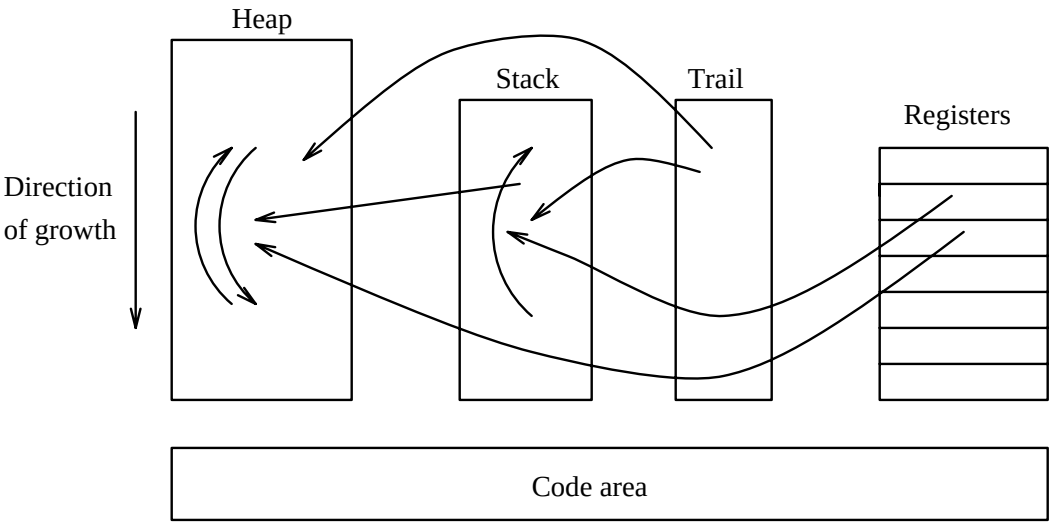


Figure 1.1: Memory layout in WAM. The arrows illustrate the directions in which references are allowed.

restore the current state at a later point, e.g., if the proof using the chosen clause fails.

To be able to restore the current state, all changes made to it have to be recorded. The only changes that occur, apart from changes to the parameters saved in the choice point, are that variables become bound. Therefore all bindings that might have to be undone are recorded on a push down list called the *trail*. Such bindings are called *conditional*.

In WAM, pointers to the arguments to a predicate are stored in a collection of argument registers, named A1, A2, etc. The first argument is stored in register A1, the second in register A2, etc.

A clause `p :- q,r,s` can be seen as a procedure `p` containing procedure calls `q,r,s`. Some registers need to be preserved across calls, e.g., registers representing variables appearing in more than one goal in the body of a clause. These registers are stored in an environment which is created on the stack (see below) when a clause is entered. The return information (the continuation) is also stored in the environment.

Instruction Set

The instruction set can be divided into four categories: instructions to choose which clauses of a predicate to try, instructions to perform the head unification,

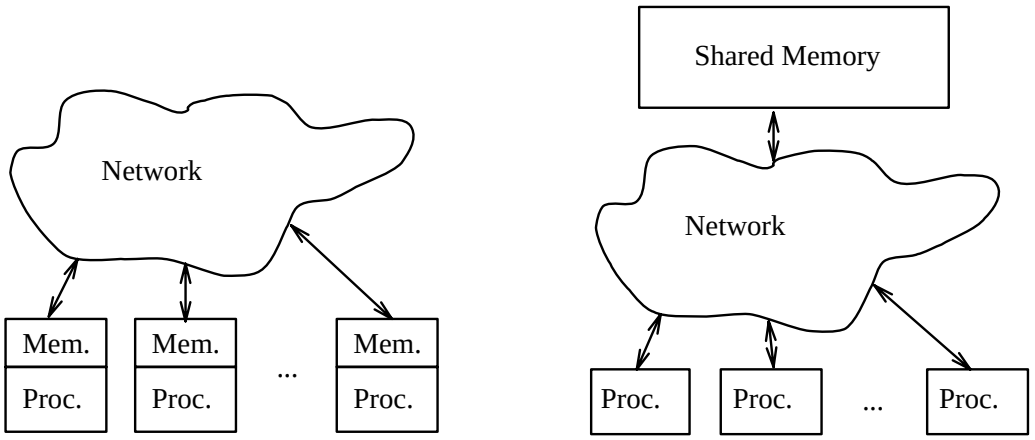


Figure 1.2: Distributed-memory multiprocessor and shared memory multiprocessor.

instruction to set up the arguments for the goals in the body of a clause, and instructions to execute the goals in the body of the clause.

Data Areas

A standard WAM implementation has three main data areas: heap, stack and trail. The heap contains non-local variables and data structures, the stack contains choice points and environments, and the trail contains references to conditionally bound variables. Environments contain housekeeping information and variables, which are either unbound, pointing to older variables in the stack, or pointing to variables or structures in the heap.

There is also a set of internal registers. The internal registers are used to keep track of the various data areas, the current program counter, where to continue after the current predicate, etc. Figure 1.1 illustrates the memory layout of the WAM.

1.3 MULTIPROCESSORS VERSUS MULTICOMPUTERS

In 1990, when we first began to consider implementing the Reform execution model, we started out with a distributed-memory architecture in mind. In September 1991 we went to the Edinburgh Parallel Computer Center and tested our implementation on their Transputer computing surface. This is a distributed memory multicomputer where all communication between processors have to be explicitly programmed, see Figure 1.2. Testing our implementation on this system was a very discouraging experience. After three weeks we had our implementation running on 17 out of 256 processors. We never managed to get the program loaded on

more than 17 processors; merely distributing the program to these 17 processors required more than 2 hours. This, in combination with the erratic behaviour of the operating system, made us look for a more attractive machine to work on. However, the experiments in Edinburgh taught us the significance of the communication speed between processing elements.

This experience made us turn our attention towards shared memory architectures, see Figure 1.2. At this time Sun released a shared memory multiprocessor, the Sun 630 MP. Our department acquired a 4 processor version of this machine. We were also given access to a 26 processor Sequent Symmetry, which is a cache-coherent bus-based shared memory machine. All our experiments have been performed on these machines.

EXECUTION MODEL

IN this chapter we describe the execution model for Reform Prolog, our recursion parallel Prolog implementation. The execution model preserves the sequential semantics of Prolog, and allows some goals to be executed in parallel. It also makes it possible for the Prolog compiler to handle large parts of the parallelization work; the runtime machinery can remain small and simple. The model is derived from the methods for recursion parallel compilation and execution proposed by Millroth [31], which in its turn is based on Tärnlund's [49] Reform inference system.

The execution model for Reform Prolog was designed in cooperation with Thomas Lindgren and Håkan Millroth.

2.1 NOTATION

Consider a recursive predicate (below) that recurs over its argument. Φ and Ψ are, possibly empty, conjunctions of goals. The nth *recursion level* refers to the goals added to the resolvent, during the nth SLD-resolution step with `p(T)` as selected atom.

```
p([]).
p([H|T]) :- Φ, p(T), Ψ.
```

This program with the query `p([a,b,c])` will produce recursion levels 1 to 3, illustrated below.

n	Recursion level	Resolvent after n resolution steps
		p`([a,b,c])`
1:	Φ, Ψ	Φ, p`([b,c])`$, \Psi$
2:	Φ', Ψ'	Φ, Φ', p`([c])`$, \Psi', \Psi$
3:	Φ'', Ψ''	Φ, Φ', Φ'', p`([])`$, \Psi'', \Psi', \Psi$

The argument being the main recursion argument in a predicate is referred to as the *recursion argument*.

2.2 REFORM COMPILATION

The idea is to compile a recursive predicate in such a way that the recursion levels can be computed in parallel. Consider the following program.

```
map([],[]).
map([X|Xs],[Y|Ys]) :- f(X,Y), map(Xs,Ys).

f(X,Y) :- Y is X - 1.
```

Given that the first argument is known to be a list of length 3 at run time, we can unfold the second clause of map/2 into the following clause.

```
map([X1,X2,X3|Xs],[Y1,Y2,Y3|Ys]) :-
    f(X1,Y1),
    f(X2,Y2),
    f(X3,Y3),
    map(Xs,Ys).
```

The three recursion levels (the calls to f/2) in this clause can be performed in parallel. Consider a recursive clause that can be written on the following form.

$$\begin{array}{lcl} p(\bar{x}) & \leftarrow & \Delta \\ p(\bar{x}) & \leftarrow & \Phi, p(\bar{x}'), \Psi \end{array}$$

If a goal $p(\bar{y})$ is determined to recurse at least n times, then the second clause of p/2 can be unfolded n times resulting in the following clause.

$$p(\bar{x}) \leftarrow \Phi_1, \cdots, \Phi_n, p(\bar{y}), \Psi_n, \cdots, \Psi_1.$$

This clause is then executed by first running the $\Phi_1 \cdots \Phi_n$ goals in parallel, then executing $p(\bar{y})$ (usually the base case), and finally running the $\Psi_n \cdots \Psi_1$ goals in parallel.

In practice the unfolded clause is never actually constructed, instead the head unification is performed at the same time as the size of the recursion argument is determined, and the body of the unfolded clause is compiled into iterative or parallel code.

When invoking a recursive program with a call of size n (corresponding to a recursion depth n) a four-phase computation is initiated:

1. A big head unification, corresponding to the n small head unification with normal control-flow, is performed.
2. All n instances of the calls to the *left* of the recursive call are computed in parallel.

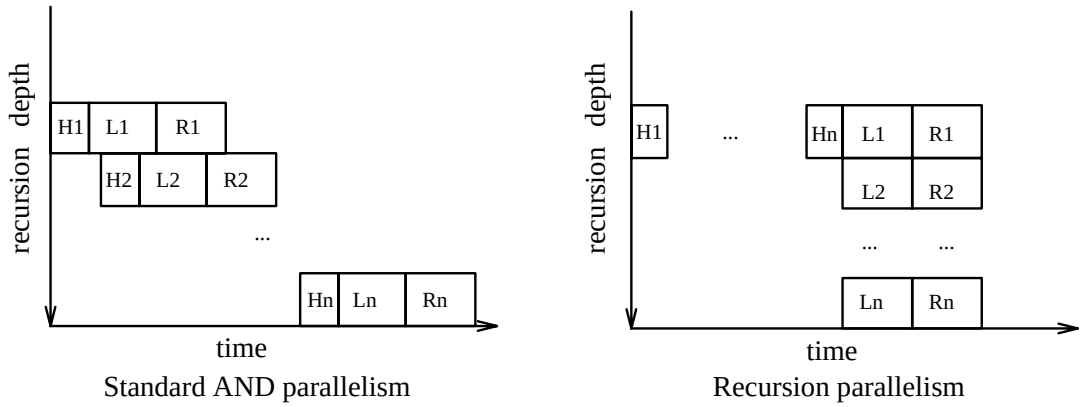


Figure 2.1: Executing a clause $H \leftarrow L, H', R$ with standard AND-parallelism and recursion parallelism.

3. The program is called recursively. This call is known to match the base clause. Hence, in practice, this call is often trivially cheap.

4. All n instances of the calls to the *right* of the recursive call are computed in parallel.

The difference between standard AND-parallelism and recursion parallelism is illustrated in Figure 2.1. The figure shows execution of a recursive clause $H \leftarrow L, H', R$, where H is the head, H' is the recursive call and L, R are (possibly empty) conjunctions. Note that the figure shows a situation where there are no data dependencies between recursion levels, and no unification parallelism.

2.3 REFORM PROLOG

In order to implement this model efficiently we have imposed a few restrictions on programs. Predicates satisfying (and predicates that can be rewritten to satisfy) the following conditions are considered for parallel execution.

1. The predicate performs integer-recursion or list-recursion, with some restrictions. Millroth [31, 34] describes these restrictions in detail. Essentially the size of the recursion has to be fixed by the time it is initiated, and the recursive argument must be reduced by a fixed amount in each recursive call.

2. All bindings of variables shared between recursion levels in the predicate are deterministic when executed sequentially. This is similar to Naish's [36] definition of binding-determinism.

3. The predicate is the topmost predicate in the call tree considered for recursion parallel execution. Nested recursion parallelism is prohibited.

Our implementation parallelizes integer and list recursion. It is straightforward to extend our scheme for many other data structures. Millroth [33] describes how this can be done for binary trees.

Note that non-determinism is allowed within each recursion level, although all bindings of shared variables must be deterministic when the entire program is executed sequentially. If conditional bindings can be created when the program is executed in parallel, e.g., if there are data dependencies between recursion levels, then suspension primitives are inserted by the compiler in order to enforce binding-determinism. The following program is binding-deterministic when executed sequentially, given the goal `p([1,1],a,F)`.

```
p([],Y,Y).
p([H|T], In, Final) :- q(In, Out), p(T, Out, Final).

q(a,b).
q(b,c).
```

However, if the program is executed using recursion parallelism, it will generate the following recursion levels.

```
q(a,Out)
q(Out,Out')
```

If the second recursion level is allowed to execute before the first recursion level has bound `Out` to `b`, then `Out` will be conditionally bound to `a`. When the first recursion level binds `Out` to `b`, the second recursion level has to be restarted. It is possible to allow this, but the machinery required for executing programs in this way is much more complicated, and has only been implemented with large overheads compared with sequential implementations. Shen [40] has proposed a machine for dependent AND-parallel execution in which the above is possible.

Suspension primitives also guarantee that sequential Prolog semantics are preserved. Suppose we have the following program and the goal `sum([1,2,3], 0, S)`.

```
sum([],X,X).
sum([X|Xs],P1,S) :- P2 is X + P1, sum(Xs,P2,S).
```

This will produce the following recursion levels.

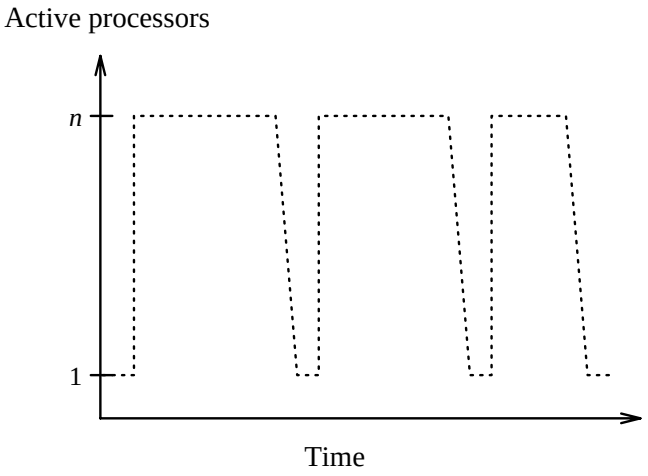


Figure 2.2: Typical processor activity during execution of a program with three consecutive calls to parallel predicates, using n processors.

```
P2 is 1 + P1
P3 is 2 + P2
P4 is 3 + P3
```

If these recursion levels were to be executed in parallel without restrictions, arithmetic errors (due to uninstantiated variables) might result. This is dealt with by inserting suspension instructions that suspend the execution of each recursion level until Pi is ground. Type testing primitives are dealt with in the same way, when necessary.

The second requirement in combination with the sequential semantics of Prolog is enough to ensure that deadlock will never occur—two recursion level cannot be mutually suspended.

The third condition ensures that only one parallel computation at a time takes place, i.e., there are never recursion parallel computations within a recursive parallel computation. If it turns out that there is not enough parallelism available from a single recursive predicate, then this restriction could be removed. That would, however, complicate both scheduling and process management. We have not investigated this further. Execution of a program alternates between phases of sequential and parallel execution. Figure 2.2 illustrates this in terms of active processors during execution of a program.

Note that it is preferable to parallelize predicates close to the top of the program. This maximizes the grain size and minimizes the sequential part of the computation. Only parallelizing single predicates deep down in a program makes it difficult

to achieve good speed-up, with the possible exception of such predicates that, by themselves, occupy a dominating part of the execution time.

Clearly advanced program analyzing techniques have to be used when compiling these programs. Such techniques are being developed by Lindgren [24].

DATA SHARING

THE idea underlying recursion parallelism is to execute every recursion level of some recursive predicates in parallel. This requires the engine executing a recursion level to have access to some part of the corresponding recursive argument, and possibly also to data produced by earlier recursion levels. In this chapter we describe how sharing of data between parallel engines running recursion levels is implemented. There are a number of problems related to this data sharing. We present our solutions to these problems, and try to motivate why we have chosen one solution before others.

3.1 MEMORY LAYOUT

The parallel abstract machine consists of a set of *workers*; each worker is a WAM-based Prolog engine. Each recursion level represents one unit of work, and is executed by a single worker. Several workers are active in parallel, executing different recursion levels. Sharing of data between recursion levels executed by the same worker is trivial, while sharing of data between recursion levels executed by different workers is an interesting problem.

The easiest way to solve this problem is to let all workers have shared access to each other's data areas. This access should be restricted through the designed of the abstract machine so that no worker may create new objects in another worker's data areas, but only bind existing variables.

In WAM there are two areas in which the representation of terms are stored: the heap and the stack. Of these, only the heap has to be shared. The stacks can remain unshared provided it can be ensured that no references to variables stored in the stack will occur in shared terms. Since, in WAM, variables stored in the heap may not point to variables stored in the stack (see Figure 1.1), the only restriction that has to be imposed is to disallow stack variables as arguments to predicates executed in parallel.

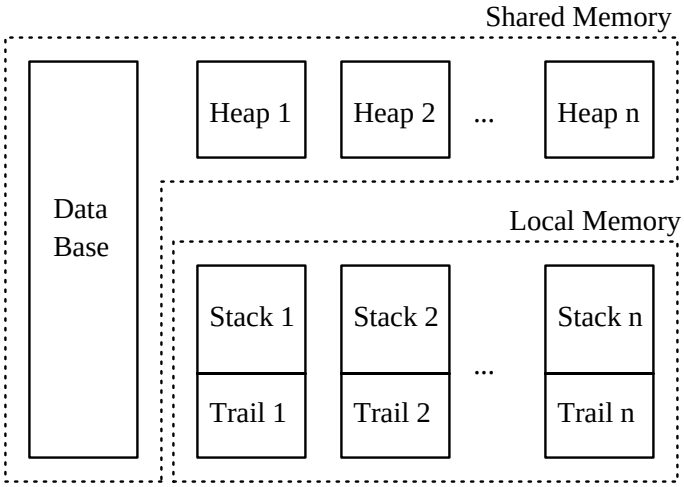


Figure 3.1: Overview of memory layout in the extended WAM.

In WAM, conditional bindings of variables are recorded on the trail, so that they can be undone on backtracking. There is no need to share this area between workers if each worker is itself responsible for undoing all conditional bindings it creates when backtracking occurs.

To summarize: Each worker has its own distinct data areas (heap, stack and trail). The stack and the trail are local to each worker and cannot be accessed by other workers. All heaps are shared and all workers have restricted access to other workers' heaps. See Figure 3.1

3.2 BINDING VARIABLES

The memory layout gives each worker the ability to refer to objects shared by other workers. It is possible for a worker to bind variables residing in other workers' heaps. This ability might lead to a situation where two workers simultaneously try to bind the same variable, resulting in one binding destroying the other. We have tried the following two methods to avoid this.

1. Lock a variable that is to be bound so that only one worker at a time has the right to bind the variable. That can be implemented using an atomic exchange operation for replacing the tag of the variable the worker wishes to bind by a lock tag. If the exchange operation returns a locked variable then another worker has already managed to lock the variable and the first worker has to wait until the lock is dropped and then try again. Such an arrangement is called a spin lock [30]. Only the variable the worker wants to

bind is locked, and there is no possibility of deadlock. The locking operation is illustrated using C code.

```
TAGGED grab_lock(term)
  TAGGED term;
{
  TAGGED result, lock = Tagify(term,LCK);

  do {
    if((result = exchange(term,lock)) != lock)
      break;
    while(*term == lock);
  } while(1);

  return result;
}
```

2. Use an atomic exchange operation when binding a variable. If another worker has already managed to bind the variable ahead of this worker, then the exchange operation will return the value to which it is bound. The other worker's binding value must then be unified with this worker's, to ensure consistency. A similar method is used in the implementation of Parallel NU-Prolog, as described by Naish [36], and in the implementation of Parallel Parlog, as described by Crammond [14]. The operation is illustrated below using C code.

```
BOOL bind(var,term)
    TAGGED var, term;
{
  TAGGED new_term;

  if((new_term = exchange(var,term)) != var)
    return unify(new_term, term);
  else
    return TRUE;
}
```

The first method has the disadvantage that it requires an extra tag. The second method is also significantly faster, as shown in section 3.5.

3.3 CREATING SHARED STRUCTURES

When building a structure on the heap, other workers should not have access to the structure until it has been fully initialized. In WAM a structure is built

using either put instructions or get instructions. When put instructions are used there is no problem, since no variable is bound to the structure until it has been fully initialized. A get instruction, on the other hand, might bind a variable to an incomplete structure and then proceed to fill in the missing parts using unify instructions. One way of avoiding references to uninitialized structures is to modify the get instructions so that they do not bind the variable, but save it for later binding. A new instruction has to be introduced after the last unify instruction (when the structure is complete) to bind the variable. This method is also discussed by Naish [36]. It is the method currently used in Reform Prolog.

There are schemas for head unification in which the transaction from read to write mode is more explicit than in WAM, e.g., the schema presented by Demoen and Mariën [28]. Our modifications for creating shared structures should be even simpler when using such schemas.

Shen [41] uses a different solution. Instead of delaying unification of an incomplete structure, the top element of the heap is marked. Other workers inspecting the structure recognize this marker and wait until it has been replaced by a value. We consider this solution inefficient, since the marker has to be written each time the top of the heap is moved. Our solution is optimized for fast building of structures and infrequent sharing of structures between workers.

3.4 CONDITIONAL BINDINGS

Conditional bindings might occur during a parallel phase even though only binding deterministic programs (with respect to shared variables) are parallelized. There are two situations in which this might happen.

1. If there is local non-determinism, then variables local to the recursion level might be conditionally bound.
2. If the parallel phase begins during nondeterministic execution, then shared variables might be conditionally bound, since the execution might backtrack to a point prior to the parallel phase.

In WAM, variables are created in chronological order on the heap. When a choice point is created, the current heap top is saved and used later for determining whether a given binding is conditional or not. Bindings of variables created before the heap top was saved are conditional. Now, if there are several heaps, as is the case in the Reform engine, then it is no longer possible to have a single saved heap top value for determining whether a binding is conditional or not.

The only way to continue using a single saved heap top value would be somehow to arrange that the heap space used by each recursion level is allocated in sequence

on a single shared heap. But since it cannot, in general, be determined in advance how much heap space a recursion level will use, this is not a feasible approach.

We have considered two solutions. Either the representation of variables is extended with a timestamp, or *all* bindings are recorded on the trail. An evaluation of both approaches are presented in Section 3.5.

Timestamped Variables

Instead of using the heap position of a variable to determine whether it was created before or after a choice point, a timestamp can be associated with each variable, and used for the same purpose. The engine must then be extended with a (time) counter which is incremented on creation of each choice point. (How this is implemented in parallel is discussed in Section 5.2.) This counter has to be saved in the choice point and restored on backtracking. The timestamp of a variable, in combination with the saved counter, can then be used for determining whether a binding is conditional or not.

The advantage of this approach, compared with recording every binding, is that the trail does not grow as fast and that unwinding the trail (undoing the bindings recorded on the trail) takes no extra time.

There are at least two ways in which a timestamp can be associated with a variable. Either the representation of *every* variable is extended with a timestamp, or the representation of *unbound* variables is modified to include the timestamp. We have tried both methods. When implementing the first method we extended the representation of variables with a timestamp having the same size as the original variable representation, which means that a variable in this representation takes 8 bytes instead of 4. This has the disadvantage that the heap consumption is nearly doubled. A slight recompense is that a larger variable size reduces the amount of false sharing. (False sharing is what happens when two processors use adjacent memory locations that reside in the same cache datum. For example, suppose byte 1 and 2 belong to the same cache datum, processor 1 use byte 1 and processor 2 use byte 2. The cache datum will then be sent between the two processors even though there is no real sharing between them.)

For the second method we introduced a new tag for unbound variables. Traditionally, an unbound variable is represented in WAM by a memory location containing a self reference. Instead we use a distinct tag for unbound variables (an *unbound tag*) which means that the value field becomes available to hold the timestamp. With this schema no extra heap space is consumed; however, when recording conditional bindings, not only the location of the variable has to be saved, but also its timestamp. The consequence is that more space is consumed on the trail, but only for conditionally bound variables; deterministic programs will use no extra space.

Using an unbound tag is no novelty, e.g., Clocksin [12] has suggested its use. Shen [41] uses an analogous method for timestamping variables in his implementation of dependent AND-parallel Prolog. Mannila and Ukkonen [27] also describe a similar notion of timestamping terms. They use the timestamp information for implementing intelligent backtracking and for fast high-level tracing of Prolog execution.

A positive side effect of using timestamps, or trailing all bindings, is that it is possible to use a copying garbage collector. We have not investigated this further, but it might be considered in future work.

Recording All Variable Bindings on the Trail

When recording every variable binding on the trail, extra space is consumed only for variables that are initially unbound and later become bound. It has the side effect that all trailing tests can be eliminated. On the other hand, since all variable bindings are recorded on the trail, backtracking becomes much more expensive.

Our experiments show that this method is much more time consuming than the methods that use timestamps.

3.5 EVALUATION

In this section we present the results of measuring the overhead of the various methods discussed earlier in this chapter.

Binding

We have measured both the time required to dereference a variable, and the time to 'dereference and bind' a variable. We measured the time for making 5 million such operations, varying the length of the variable chain from 1 to 3. Touati and Despain [47] show that most (an average of more than 95%) variable chains are shorter than 3. The following times were measured on a 4 processor Sun 630 MP and a 26 processor Sequent Symmetry.

	Unmodified	Using Spin Lock		Using Atomic Bind	
Program	Time (s)	Time (s)	Overhead	Time (s)	Overhead
Sun 630 MP					
Deref (1)	1.63	1.75	7%	1.63	0%
Deref (2)	3.00	3.50	17%	3.00	0%
Deref (3)	4.38	4.88	11%	4.38	0%
Deref + bind (1)	2.25	3.63	61%	2.75	22%
Deref + bind (2)	3.63	4.90	34%	4.12	13%
Deref + bind (3)	5.00	6.14	23%	5.50	10%

Program	Unmodified Time (s)	Using Spin Lock Time (s)	Using Spin Lock Overhead	Using Atomic Bind Time (s)	Using Atomic Bind Overhead
Sequent Symmetry					
Deref (1)	16.67	16.68	0%	16.67	0%
Deref (2)	28.64	28.70	0%	28.64	0%
Deref (3)	39.53	39.96	0%	39.53	0%
Deref + bind (1)	16.69	32.41	94%	21.40	28%
Deref + bind (2)	29.27	45.30	54%	36.82	26%
Deref + bind (3)	40.59	57.89	40%	51.97	28%

It is clear that atomic bind is preferable.

Trailing

These tests do not take false sharing into account. Reduction of false sharing is achieved when using timestamps. Frequent false sharing slows down execution considerably.

We have measured the execution time for 5 million operations related to trailing on a Sun 630 MP and a Sequent Symmetry.

	Unmod.	T.stamp 1		T.stamp 2		T. all	
Sun 630 MP							
init	0.75	1.36	81%	0.75	0%	0.75	0%
trail1	3.13	3.38	8%	4.75	52%	2.26	-33%
trail2	1.00	1.25	25%	1.23	23%	2.26	126%
trail3	1.00	1.25	25%	1.23	23%	6.30	530%
Sequent Symmetry							
init	5.66	10.07	78%	5.04	-11%	5.66	0%
trail1	14.79	16.36	11%	22.97	55%	11.97	-19%
trail2	11.32	12.89	14%	11.64	3%	11.97	6%
trail3	11.32	12.89	14%	11.64	3%	36.41	222%

The measured times are in walltime seconds. In the table above *init* refers to initalizing a variable, *trail1* refers to trailing a conditional binding, *trail2* refers to trailing an unconditional binding, and *trail3* refers to trailing an unconditional binding, and unwinding the trail. 'T.stamp 1' is the method where the representation of each variable is extended with a timestamp field, 'T.stamp 2' is the method using a special tag for unbound variables. 'T. all' is the method where all bindings are recorded on the trail, conditional as well as unconditional.

It is not clear which method is the faster from the figures above; one also has to consider the frequency of each operation. This is hard since it depends on the programs that are executed. We therefore measured the execution times of two full programs in the emulator. The naive reverse program is completely deterministic, while the metainterpreter is not.

Program	Unmodified	T.stamp 1	T.stamp 2	Trail All
Sun 630 MP				
metainterpreter	0.67	0.71	0.70	0.71
naive reverse	0.11	0.12	0.12	0.12
Sequent Symmetry				
metainterpreter	4.80	4.84	5.13	4.8
naive reverse	0.97	0.97	1.06	1.04

The measured times are in walltime seconds. According to these figures the choice of method makes no significant difference. Using timestamps and a special tag for unbound variables requires less space than any of the other methods, but is slightly more time consuming.

It should be noted that it is difficult to measure small differences in execution times on the Sun 4 machines. Small changes in a program might result in considerable changes in execution time, due to cache effects that are hard to predict. We have a small test program that demonstrates this behavior. The program adds two vectors, producing a third. If the vectors are allocated in sequence, then the program is slow. If, e.g., 54 bytes are allocated between the vectors, on the other hand, then it executes in about 1/3 of the time (26.3 seconds compared with 8.0 seconds). The reason for this appears to be that Sun 4 machines use directly mapped caches.

Process Management

Process management and scheduling are critical points in many of the parallel Prolog systems existing today. OR-parallel systems, such as Aurora [9] and Muse [20], depend heavily on scheduling algorithms to distribute work among workers. In AND and OR parallel systems it is essential not to exploit all possible parallelism since the scheduling and process spawning overhead might easily grow out of hand. Granularity control is a challenging problem, and a variety of solutions have been proposed [19, 21].

Whether to do speculative work or not, is also a delicate problem in these systems. An example of speculative work in an OR-parallel system is to execute a branch of the search tree that might be cut away by another branch to the left of it. A similar problem exists in most AND-parallel systems, it occurs when goals are performed to the right (in the resolvent) of a goal that might fail.

Another problem, common to both AND-parallel and OR-parallel systems, is that all parallel processes cannot be initiated at the same time. Instead they are spawned from other processes during the parallel execution, which results in a dynamic process network. It is also impossible, in these systems, to know in advance which code each process is going to execute.

In recursion parallel systems much of the scheduling is done at compile time. It is then possible to determine which code is going to be executed in parallel. The number of processes executing the parallel code is determined immediately before the parallel phase is initiated. These properties significantly simplify the process management problem. In fact, the scheduling overhead becomes negligible. Note that this implementation of recursion parallelism is optimized for running recursion parallel programs where every parallelized recursion level succeeds.

4.1 SUSPENSION

The set of programs that can be parallelized in Reform Prolog has been restricted to those which are binding-deterministic with respect to shared variables. This condition is verified at compile time. Some programs require runtime tests to ensure that the sequential semantics of the program is preserved. The following runtime conditions for continued forward execution are generated by the compiler when needed.

1. Is a certain variable instantiated? If not, suspend until it becomes instantiated. If the goal becomes leftmost and the variable remains uninstantiated, then signal a runtime error.
2. Is the goal leftmost? If not, suspend until it becomes leftmost.
3. Is a certain variable instantiated *or* is the goal leftmost in the resolvent? If not, suspend until either becomes true.
4. Is a certain variable *un*instantiated *and* is the goal leftmost? If not, suspend until the goal is leftmost. If the variable becomes instantiated during suspension, then signal a runtime error.

Performing these tests is trivial, suspending until they are satisfied is not. We have investigated two ways of implementing suspension until the conditions are satisfied: one solution uses semaphores to activate a suspended recursion level, and the other employs active polling by the suspended recursion level.

Using Semaphores

Semaphores can be used in the following way. When a test results in suspension, the suspended recursion level reads a semaphore. That semaphore should be signaled as soon as the test can be satisfied or decided never to be satisfied; either when a variable becomes bound or when the suspended recursion level becomes leftmost in the resolvent. This can be implemented using a technique similar to that used when implementing *diff* and *freeze* (as described by Carlsson [10]) in combination with a mechanism to signal the appropriate semaphore when a recursion level becomes leftmost.

The drawback of this method is that it imposes a comparatively large overhead on active recursion levels. All recursion levels are slowed down since they have to check if they should signal a semaphore.

The advantage is that while a recursion level is suspended, another recursion level could be active and use the processing power otherwise wasted by the suspended recursion level. This might, for example, be done by having more workers than

processors. For this to become a significant advantage, recursion levels must spend a fairly large amount of time suspended.

Using Polling

Using polling, a recursion level waiting for a certain condition to become true actively checks the condition until it can determine whether it is true or false. For example, verifying condition 1 above would mean repeatedly checking if a variable has become instantiated or if the goal has become leftmost.

This method has the drawback that a suspended recursion level can waste significant processing power performing the suspension check. Whether this is a problem or not depends on the programs that are run and on how sophisticated the scheduling mechanism is. The cost of verifying whether a recursion level is leftmost or not grows with the number of processors. Therefore, this method might not be suitable for massive parallelism.

The positive aspect of this method is that it does not slow down processes that do not suspend—only the waiting worker is slowed down. No extra overhead is imposed on the abstract machine compared with a sequential implementation.

4.2 SCHEDULING

One of the advantages of Reform compilation, and data-parallelism in general, is that much of the scheduling can be done at compile time. It is possible to determine which code is going to be executed in parallel, but the exact number of recursion levels can, in general, only be decided at run time; then it is given by the size of the recursion argument.

The scheduling activity is thus reduced to dividing recursion levels among workers. There are two basic approaches to scheduling: dynamic scheduling and static scheduling. There are also at least two ways of mapping recursion levels to processors. Either consecutive recursion levels are mapped to consecutive processors (*horizontal mapping*), or consecutive recursion levels are mapped to the same processor (*vertical mapping*), see Figure 4.1. If there are data dependencies in the program, then horizontal mapping is preferred since it enables overlapping. If there are no data dependencies, then it might be better to use vertical mapping since the likelihood of data locality is higher.

Static Scheduling

It is desirable to use static scheduling to minimize the need for synchronization and context switching. Most parallel Prolog systems cannot use static scheduling, since too little information is available about the structure of the parallel execution

Worker	1	2	3	4	5	6	7	8
Rec. level	1	2	3	4	5	6	7	8
	9	10	11	12	13	14	15	16
	17	18	19	20	21	22	23	24

Horizontal mapping

Worker	1	2	3	4	5	6	7	8
Rec. level	1	4	7	10	13	16	19	22
	2	5	8	11	14	17	20	23
	3	6	9	12	15	18	21	24

Vertical mapping

Figure 4.1: Vertical and horizontal mapping of recursion levels to workers.

before it is initiated. In recursion parallel systems information about which code is going to be executed, and the number of parallel processes, is available. This makes it possible to statically distribute recursion levels to workers.

Some programs are not suited for static scheduling, e.g., programs in which there are large differences between the execution times of different recursion levels.

Dynamic Scheduling

Dynamic scheduling has received much attention in the context of loop parallelization. Several more or less sophisticated methods have been proposed. The objective of these methods is to minimize synchronization and context switching, while at the same time giving a reasonable load balance. This is hard to achieve on systems where synchronization is expensive, and for programs where the execution time of each iteration is small. Small execution times for each iteration are common when parallelizing loops, since there is no general technique for parallelizing loops containing procedure calls.

The simplest dynamic algorithm for scheduling loop iterations is called *self-scheduling* [44]. In this algorithm each processor allocates and executes one iteration at a time from a central queue of iterations, until all iterations have been executed. In the case of recursion parallelism, one recursion level is allocated and performed by each processor until all recursion levels have been executed. This method achieves almost optimal load balancing. The problem with self-scheduling is that the access to the central queue of remaining iterations, or recursion levels, might become a bottleneck. On large systems where many processors must access the queue, such

a device is prone to become a bottleneck. Even on small-scale systems it might become a problem if the time to execute each recursion level is small.

Uniform-size chunking [22] reduces synchronization overhead by each time allocating a number, K, iterations to each processor. The worst case is that processors finish within K iterations of each other. It is, however, hard to decide which value to use for K, since the optimal value depends on the program being executed.

A more sophisticated algorithm is *guided self-scheduling* [38]. This algorithm allocates $1/P$ times the remaining number of iterations for each processor to execute, where P is the number of processors. The drawback of this method is that towards the end few iterations are allocated per processor.

A dynamic scheduling algorithm that preserves many of the good characteristics of static scheduling, but also achieves good load balance for programs that are not well suited for static scheduling, has been proposed by Markatos and LeBlanc [29]. The idea behind this algorithm is to use static scheduling as far as possible, but if load imbalance occurs, i.e., a processor is idle while there are iterations to be executed, then iterations migrate from one processor to another.

In AND-parallel logic programming implementations the set of parallel processes grows during parallel execution. This quality makes these implementations unsuitable for the scheduling algorithms described above. Hermenegildo [18] describes a scheduling algorithm for &-Prolog in which each processor keeps a local goal-queue. When a goal-queue becomes empty, goals migrate from other queues. Crammond [14] uses a similar scheduling algorithm in his implementation of Parlog, the main difference being that local goal-queues are devided into private and global parts. The advantage of this is that the local parts of the goal-queue can be updated without locking. Any one of these algorithms can be used for scheduling recursion levels in Reform Prolog. However, there are two reasons why such elaborate algorithms are not needed in Reform Prolog.

1. The number of parallel processes is fixed during parallel execution.
2. The execution time (the grain size) of each recursion level in Reform Prolog is, in general, much larger than the size of the goals scheduled in other AND-parallel systems. The reason for this is that Reform Prolog parallelizes selectively close to the top-level of a program, whereas other AND-parallel system parallelize smaller units of work.

4.3 EVALUATION

Static scheduling and self-scheduling have been implemented. We could not measure any difference between these two scheduling methods when running our benchmark programs. The reasons for this are either that the programs are well balanced

(all but nrev) or that data dependencies, in combination with polling suspension, eliminate the effects of load imbalance (nrev).

WAM EXTENSIONS

IN order to implement the data sharing and process management designs described in chapters 3 and 4, a number of extensions have to be introduced in WAM. The abstract machine has to be extended so that several recursion levels can be executed in parallel, sharing certain data. The shared data must be protected by synchronization so that all recursion levels have the same view of the shared data as they would have had during a sequential execution of the same program. Extensions for controlling the parallel execution, and for distributing work among processors, must be added. Most of the extensions are introduced as new instructions in WAM, but some existing mechanisms, such as dereferencing and binding, are also modified.

The instruction set described in this chapter was partly designed in cooperation with Thomas Lindgren and Håkan Millroth.

5.1 OVERVIEW

The parallel machinery consists of a set of workers numbered 0,1,...,$n-1$, one per processor. Each worker is implemented as a separate process running a WAM-based Prolog engine with extensions to support parallel execution. The execution of a program alternates between two modes: sequential execution and parallel execution. A phase of sequential execution is referred to as a *sequential phase* and a phase of parallel execution as a *parallel phase*. One worker is responsible for sequential execution (the *sequential* worker). During sequential execution all other workers (the *parallel* workers) are idle, during parallel execution the sequential worker is idle. The terms created by the sequential worker must be accessible to the parallel workers during parallel phases. This includes variables, numbers, structures, and lists. Terms created by the parallel workers must likewise be accessible to other workers during the parallel phase, and to the sequential worker during the next sequential phase. Sharing of data is implemented by letting all workers have restricted access to each others heaps, as described in Section 3.1.

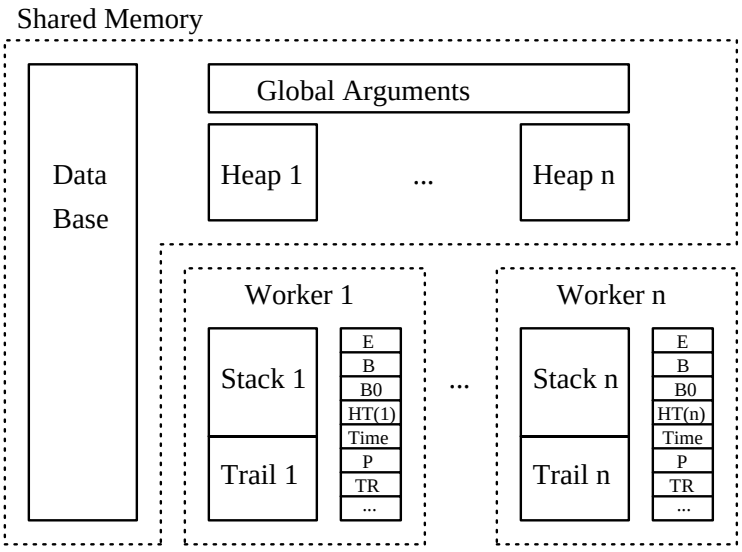


Figure 5.1: Overview of the recursion parallel machine.

It is the sequential worker's responsibility to initiate parallel execution and to resume sequential execution when the parallel workers have executed all recursion levels. The sequential worker sets up the arguments for each recursion level in its own argument registers, which are globally accessible.

Through the extensions in this chapter, a form of data-parallelism is introduced into WAM. The data-parallelism used is different from that described by, e.g., Hatcher and Quinn [16], in that the same program is executed by all workers, but not necessarily exactly the same instructions—non-determinism and different deterministic choices of clauses in predicates might result in that different parts of the same program are executed by different recursion levels. Also, barrier synchronization is used only when ending a phase of parallel execution.

5.2 DATA SHARING

The workers have a common address space which is divided into heaps. Each heap is managed by one worker, which has the right to create new objects on the heap and is responsible for garbage collection. The heaps are essentially statically allocated. They can be allowed to grow and shrink in the same way as a heap in a sequential WAM implementation.

The database is also located in shared memory, so all workers have access to it. The common *logical* database view is used, as described by Lindholm and O'Keefe [25]. Only predicates declared as being *dynamic* may be modified at run time. The logical view of the database, in combination with the restriction on which

predicates are permitted to be updated, allows us to synchronize access to the database only when reading dynamic predicates; reading other predicates can be done without synchronization. Whenever a dynamic predicate is read from the database during a parallel phase, the worker reading the predicate has to suspend until it becomes leftmost.

Binding Variables

The binding mechanism is modified as described in Section 3.2. In short, rather than overwriting the previous contents of a variable's cell, the variable's new value is exchanged for its old value, using an atomic exchange operation. The old value is examined, and if another worker has managed to bind it ahead of the present worker, then its old value is unified with the new value.

Trailing

A new tag for unbound variables and a timestamp mechanism have been introduced in order to distinguish between conditional and unconditional bindings of variables, as described in section 3.4.

Introducing a special tag for unbound variables means that we introduce tagged objects in WAM. Only two operations have to be modified, the dereferencing operation and the operation that initializes variables.

In order to use timestamps, WAM has to be extended with a counter which is incremented when choice points are created. The previous time is saved in a choice point, so that it can be restored on backtracking. When a parallel phase is initiated, all parallel workers are given the sequential worker's current time count for initializing their local time counter. This means that two parallel workers might give variables identical timestamps even though that would not have been the case during sequential execution. This is not a problem since each worker only trails conditional bindings created before the current parallel phase started, and conditional bindings of local variables. Unconditional bindings of shared variables might appear to be conditional using this schema. This might result in unnecessary trailing in some cases. After the parallel phase has terminated, the sequential worker receives all parallel workers' time counts and uses the highest as its new time count.

Note that in WAM implementations in which heap and stack variables have distinct tags, only heap variables need to be extended with a timestamp. Stack variables can be handled as in an ordinary WAM implementation.

Creating Shared Structures

In the standard WAM a variable might be bound to a structure before the structure is fully constructed, for example, by the `get_structure` instruction. This is not a satisfactory behavior in an environment where several workers can access a shared variable simultaneously. Instead of binding the variable to an uninitialized structure, the variable is saved in a register and bound to the structure once it has been fully initialized. This is implemented by introducing three new instructions: `lock_and_get_structure`, `lock_and_get_list`, and `unlock`.

5.3 SCHEDULING

When using static scheduling each worker is responsible for calculating which recursion levels it is going to execute. Before the first recursion level is executed, each worker calculates the number of its first recursion level, using its own worker number. The current recursion level is saved in an internal register, and incremented or decremented when the worker changes its recursion level. When the register is decremented below 0 (when executing the goals to the left of the recursive call), or incremented above the length of the recursion list (when executing the goals to the right of the recursive call), then the worker terminates its part of the parallel phase.

When using dynamic scheduling, a globally accessible counter is used to hold the next recursion level to be performed. A worker that is looking for work locks that counter and allocates as many recursion levels as the scheduling algorithm proclaims.

Both dynamic and static scheduling are implemented trough the `spawn_*` instructions described below. The `start_*_body` instructions are responsible for initializing the global counter when using dynamic scheduling.

Suspension

Data dependencies are allowed in the parallel code. Even though binding-determinism for shared variables is required of the parallelized programs, some form of synchronization is needed, since a program need only be binding-deterministic when executed sequentially. The compiler analyzes the program and inserts synchronization primitives in the code when needed to ensure binding-determinism. The algorithm for this is described by Lindgren [24].

There are four synchronization primitives. They verify the instantiation of a term and suspend until the object complies with the required instantiation, or until the current recursion level is leftmost in the resolvent. A suspended worker repeatedly verifies the type of the term and checks if the current recursion level has become leftmost.

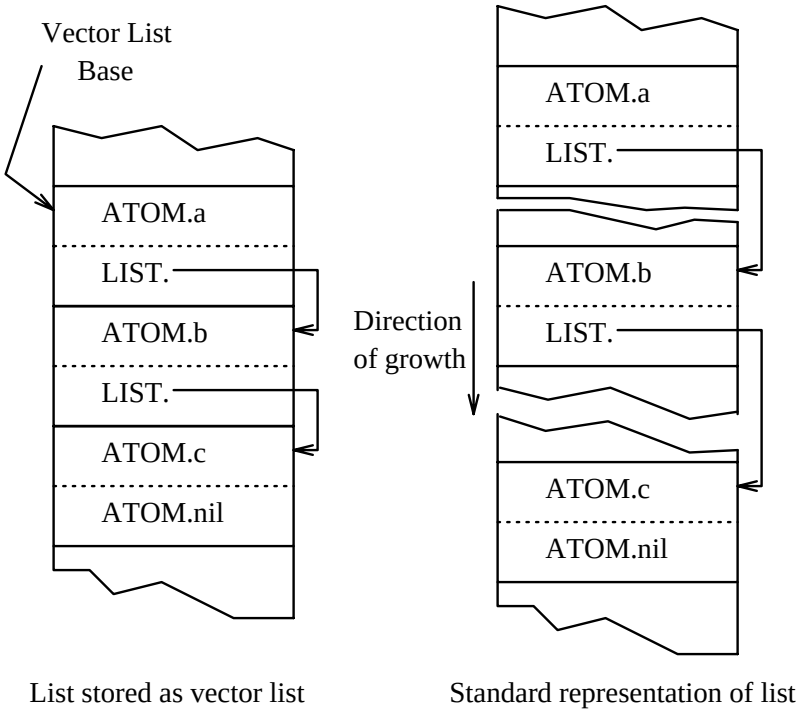


Figure 5.2: The list `[a,b,c]` stored as a vector list and in the standard way.

Each worker has been extended with a register that is accessible by all workers. This register contains the number of the recursion level that the worker is currently executing. A suspended worker continuously looks at the other workers' registers to see if it has become leftmost.

5.4 VECTOR LISTS

A recursion parallel program that performs list recursion is parallelized so that some recursion levels can be processed in parallel. Now, there are at least two ways for workers to access their elements in the recursion list. One way is giving every worker a pointer to the head of the list and letting them traverse the list until they find the elements they are looking for. This will obviously lead to much duplicated work in traversing the list. The other way, the one we have chosen, is to traverse the list once and build a vector containing its elements before initiating the parallel phase. In this way, each worker can simply index into the vector to get its particular element, without having to traverse the entire list. The problem is that when the parallel phase is finished, the vector should be viewed as a list again. Our solution is to make the vector *appear* as a list, or rather, represent the list as a vector (see Figure 5.2). We could also have chosen to implement the more

elaborate cdr-coding schema described by Millroth [32], but our method is simpler and does not require an extra tag.

Vector lists are also used for sharing variables between recursion levels. This use corresponds to the use of environments in a sequential WAM. Consider the following program.

```
sum([],S,S).
sum([X|Xs], N, S) :- N2 is X + N, sum(Xs,N2,S).
```

The first argument is transformed into a vector list for inexpensive access from all recursion levels. The third argument is not affected, since it is only passed on to the base case. Processing the second argument is an interesting problem. Each recursion level needs to refer to the number computed by the previous recursion level. In a sequential WAM this might be done by representing `N2` as an environment variable (or storing it in a register if arithmetic operations are given special treatment). This is not a feasible solution in our system since stacks are not shared between workers. Even if they were shared, it would not be possible to store the variables in the stack since they have to be created before the parallel execution is initiated.

The solution is to create on the heap a vector list containing these variables, before the parallel phase is initiated. The vector list is of length $n + 1$, where n is the number of recursion levels to be executed. The first element in the vector list is set to the initial value of the second argument. The rest of the vector list is initialized with distinct variables.

Consider the list in the first argument position of sum/3 above. The sequential worker traverses the list and builds a vector list of its elements before initiating the parallel phase. This is neccessary since the worker needs to know the total recursion depth (i.e., the length of the list) *before* the parallel phase is initiated. Otherwise it would not know how many parallel processes (recursion levels) to spawn, or the size of the other vector lists.

Vector lists are an example of the tradeoff between efficient memory usage (stack allocation) and increased potential for parallelism (heap allocation). For tail recursive programs this effect is even more dramatic.

5.5 INSTRUCTION SET

The instruction set extensions can be divided into the following five categories.

1. Instructions for building vectors: `build_*`
2. Instructions for process control, to initiate the parallel execution, and to go from one recursion level to the next: `start_*, initialize_*, spawn_*`

3. Instructions for fetching data from the sequential worker's space when executing in parallel: `put_*, unify_*`
4. Instructions for managing shared structures: `lock_and_*, unlock`.
5. Instructions for runtime tests to ensure global binding-determinism: `await_*`

We have modified the unification and the binding operation to handle potentially shared variables. These modifications are only needed during a parallel phase, and then only for variables that are shared. One might therefore consider adding specialized "local" versions of all WAM instructions that perform unification or bind variables. These instructions could be used during sequential execution and during parallel execution for variables that have been determined to be local through program analysis. We have not done this since the overheads due to the modifications to the unification and binding operations are small. In a native code implementation it would be natural to distinguish between potentially shared variables and local variables.

Note that if reductions were actually compiled as in the sum/2 example below (see page 43), the execution would be sequentialized through data dependencies—only overlapping parts of such programs would be parallelized. Reductions using addition and multiplication are instead compiled as calls to built-in predicates which utilize parallel reduction algorithms to reduce vector lists. The full compilation schema for reductions is described by Lindgren [24].

Vector List Instructions

These instructions are used to build vector lists that are used for easy access of input arguments, and for communication between recursion levels during the parallel phase. The instructions prepare the environment for the following parallel phase and are as such executed by the sequential worker. A list recursive program such as

```
map([],[]).
map([X|Xs],[Y|Ys]) :- rel(X,Y), map(Xs, Ys).
```

is compiled to extended WAM code according to the following schema:.

```
map/2:  switch_on_term Lv La Ll fail

Lv:     try La
        trust L1

La:     /* sequential code for first clause */
```

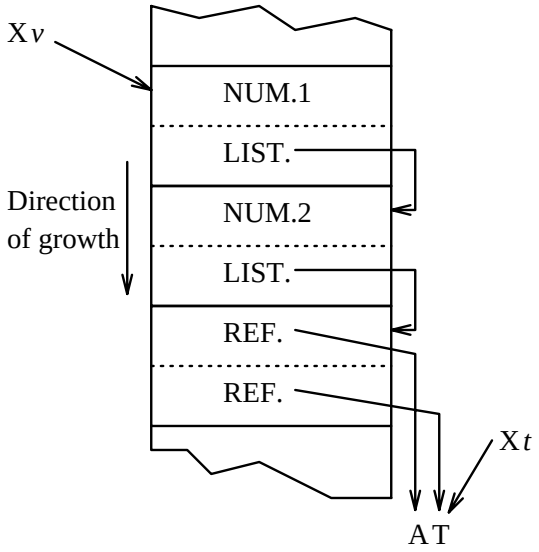


Figure 5.3: The list `[1,2,A|T]` stored as vector list created by `build_poslist` with length 3.

```
L1:     /* sequential code for second clause */

Ll:     build_rec_poslist X0 X2 X3 X0  % create vector list Xs
        build_poslist X1 X2 X4 X1      % create vector list Ys
        /* code to execute rel/2 by each worker */
        execute map/2                  % execute the base case
```

The compilation schema is described in detail by Lindgren [24].

`build_rec_poslist`(*a, n, v, t*)
: This instruction is used for the main recursive argument when compiling list recursion, e.g, the first argument of map/2 above. It verifies that the term referred to by $\mathtt{X}_a$ is a list. The list is traversed and a vector list is built (on the heap) of its elements. A pointer to the vector list is stored in $\mathtt{X}_v$. The length of the vector list is stored in $\mathtt{X}_n$ (it is used later when constructing other vector lists and by each worker to determine the bounds of the recursion).

`build_poslist`(*a, n, v, t*)
: This instruction is used for an argument that performs list recursion but is not the main recursive argument, e.g., the second argument of map/2 above. The term pointed to by $\mathtt{X}_a$ is verified to be a list of length at lest $\mathtt{X}_n$. If $\mathtt{X}_a$ contains a list shorter than $\mathtt{X}_n$ elements, ending with an unbound variable, then it is extended with distinct variables to the length $\mathtt{X}_n$. A vector list is constructed (on the heap) of its $\mathtt{X}_n$ first elements. A pointer to the vector

list is stored in X_v. The X_nth tail of the vector list is set to the X_nth tail of the list referred to by X_a. A reference to the X_nth tail is also stored in X_t. (This reference is needed when executing the base case of the predicate. See Figure 5.3.)

`build_poslist_value`(*a, n, v, t*)
: This instruction is used for a subsequent argument that performs list recursion, using the same element as an earlier list recusive argument, e.g., the third argument of the second clause of append/3.

```
append([X|Xs],Y,[X|Zs]) :- append(Xs,Y,Zs).
```

The term referred to by X_a is verified to be a list. The first X_n elements of the list X_a are unified with the corresponding elements of the vector list referred to by X_v. If the list X_a is shorter than X_n elements then the last tail of the list is unified with a list of the remaining elements of the vector list, ending with an unbound variable. The X_nth tail of X_a is stored in X_t.

`build_neglist`(*a, n, v, t*)
: This instruction is used for a list that is built in the clause body using an accumulator, e.g., the second argument of the predicate below.

```
p([X|Xs],Acc,[Y|Ys]) :- p(Xs,[Y|Acc],Ys).
```

A 'reverse list' vector list of length X_n is created on the heap. A reference to the vector list is stored in X_v. A reference to the first element of the vector list (viewed as a list) is stored in X_t. The last tail of the vector list (viewed as a list) is set to X_a. See Figure 5.4.

`build_neglist_value`(*a, n, v, w, t*)
: This instruction is used for a subsequent list that is built (using the same element as an earlier list argument) in the clause body using an accumulator, e.g., the second argument of reverse/3 below.

```
reverse([X|Xs],Acc,Rev) :- reverse(Xs,[X|Acc],Rev).
```

The instruction, analogous to `build neglist`, creates a 'reverse list' vector list X_w, but elements from the vector list referred to by X_v are stored at the corresponding vector list positions in X_w.

`build_variables`(*a, n, v, t*)
: This instruction is used for a variable argument that is not passed on to the next recursion level; a new term takes its place, e.g., the second argument of sum/3.

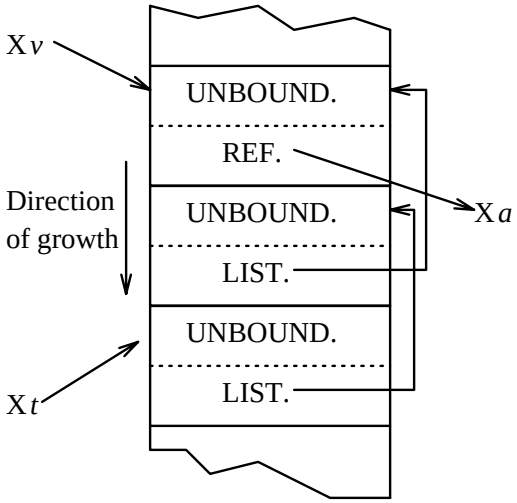


Figure 5.4: List (viewed as a vector) created by `build_neglist`.

```
sum([X|Xs],N,S) :- N2 is X + N, sum(Xs,N2,S).
```

Let N be the contents of $\mathtt{X}_n$ and A the contents of $\mathtt{X}_a$. The instruction creates a vector list of length $N+1$, storing a pointer to it in $\mathtt{X}_v$. The first element of the vector list is set to A; all other elements are distinct unbound variables. A reference to the $N+$ 1st element of the vector list is stored in $\mathtt{X}_t$.

Procedural Instructions

These instructions are used to initiate parallel execution and to switch from one recursion level to the next. The `spawn` instructions corresponds to the `while` instruction of C. The program

```
p([],Y,Y,0).
p([X|Xs], Ys, Zs, S1) :-
        f(X, Y),
        p(Xs, [Y|Ys], Zs, S2),
        q(S2, S1).
```

is compiled to extended WAM code according to the following schema:.

```
p/4:    switch_on_term Lv La Ll fail

Lv:     try La
        trust L1
```

```
La:     /* sequential code for first clause */

L1:     /* sequential code for second clause */

L1:     allocate
        build_rec_poslist X0 X4 X5 X0  % create vector list Xs
        build_neglist X1 X4 X6 X1      % create vector list Ys
        build_variables X3 X4 X7 X3    % create var. vector list
        start_left_body L2             % start par. phase
        /* code to save X4, X5, X6, X7 in Y-registers */
        call p/4, 4                    % execute base case
        /* code to restore X4, X5, X6, X7 from Y-registers */
        start_right_body X4 L4         % start par. phase
        deallocate
        proceed

L2:     initialize_left 1              % initialize worker
L3:     spawn_left 1 X2 G4             % while ++X2 < G4 do
        /* code to set up the arguments for the call to f/2 */
        call f/2, 0
        jump L3                        % next recursion level

L4:     initialize_right 1 G4          % initialize worker
L5:     spawn_right 1 X2               % while --X2 > 0 do
        /* code to set up the arguments for the call to q/2 */
        call q/2, 0
        jump L5                        % next recursion level
```

Step is the number by which each recursion level decrements the recursive argument. For example, the step of `append/3` is 1 since one element is removed from the list by each recursive call.

`start_right_body`(*n*, *L*)
: This instruction initiates parallel execution of the 'right body' of a predicate. The length of the recursion list is given in $\texttt{X}_n$. The code at label *L* is run in parallel by all active workers. The sequential execution continues with the next instruction when the parallel phase is finished.

`start_left_body`(*L*)
: This instruction starts parallel execution of the 'left body' of a predicate. The code at label *L* is run in parallel by all active workers. The sequential execution continues with the next instruction when the parallel phase is finished.

`initialize_right`(*S*, *n*)
: This instruction initializes a worker for parallel execution. The step *S*, the

number of recursion levels (stored in the sequential worker's register $\texttt{X}_n$), and the worker number are used for calculating the initial recursion level in static schedule mode. When dynamic scheduling is used this instruction is ignored.

`initialize_left`(S)
: This instruction initializes a worker for parallel execution. The step S and the worker number is used for calculating the number of the initial recursion level in static scheduling mode. In dynamic scheduling mode this instruction is ignored.

`spawn_right`(S, i)
: This instruction calculates the number of the next recursion level and stores it in $\texttt{X}_i$. The new level is calculated from S and the internal level count. If the number of the new level is negative, the parallel computation is finished and the worker awaits the next parallel phase, otherwise processing continues with the next instruction.

`spawn_left`(S, i, n)
: This instruction calculates the number of the next recursion level and stores it in $\texttt{X}_i$. The number of the new level is calculated from S and the internal level count. If the number of the new level is greater than the value stored in the sequential worker's register $\texttt{X}_n$, the parallel computation is finished and the worker awaits the next parallel phase, otherwise processing continues with the next instruction.

Global Argument Instructions

These instructions are used by each worker in the parallel phase to fetch the arguments of the recursion level from the sequential worker's heap. The program

```
map([],[]).
map([X|Xs],[Y|Ys]) :- rel(X,Y), map(Xs,Ys).
```

is compiled to extended WAM code according to the following schema:

```
map/2:  switch_on_term Lv La Ll fail

Lv:     try La
        trust L1

La:     /* code for first clause */

L1:     /* code for second clause */
```

```
L1:     build_rec_poslist X0 X2 X3 X0 % build vector list Xs
        build_poslist X0 X2 X4 X1     % build vector list Ys
        start_left_body L2            % start par. phase
        execute map/2                 % execute base case

L2:     initialize_left 1             % initialize worker
L3:     spawn_left 1 X2 G2            % while X2 < G2 do
        put_nth_head G3 X2 0 X0       % X <- Xs[X2+0]
        put_nth_head G4 X2 0 X1       % Y <- Ys[X2+0]
        call rel/2 0                  % rel(X,Y)
        jump L3                       % next rec. level
```

`put_nth_head(`*v, l, Offset, i*`)`
: This instruction stores element number $\mathtt{X}_l$ + *Offset* of the vector list, pointed to by the sequential worker's register $\mathtt{X}_v$, in $\mathtt{X}_i$. Nonzero values of *Offset* are used by clauses that reduce the recursion argument by more than 1, or by clauses that need to refer to values produced by other recursion levels.

`put_nth_tail(`*v, l, Offset, i*`)`
: This instruction stores tail number $\mathtt{X}_l$ + *Offset* of the vector list, pointed to by the sequential worker's register $\mathtt{X}_v$, in $\mathtt{X}_i$.

`put_global_arg(`*g, i*`)`
: This instruction stores the value of the sequential worker's register $\mathtt{X}_g$ in $\mathtt{X}_i$.

`unify_nth_head(`*v, l, Offset*`)`
: This instruction writes element number $\mathtt{X}_l$ + *Offset* of the vector list, pointed to by the sequential worker's register $\mathtt{X}_v$, onto the heap. This instruction never occurs in read mode.

`unify_nth_tail(`*v, l, Offset*`)`
: This instruction writes tail number $\mathtt{X}_l$ + *Offset* of the vector list, pointed to by the sequential worker's register $\mathtt{X}_v$, (viewed as a list) on the heap. This instruction never occurs in read mode.

`unify_global_arg(`*g*`)`
: This instruction writes the value of the sequential worker's register $\mathtt{X}_g$ on the heap. This instruction never occurs in read mode.

Shared Structure Instructions

These instructions are used when unifying structures in the head that may be shared with other workers. Instead of binding a variable to an uninitialized structure, the variable is saved in a register and bound to the structure when the

structure has been fully initialized. The following program illustrates the use of these instructions (calls to nrev/2 are parallelized).

```
nrev([],[]).
nrev([X|Xs], Y) :- nrev(Xs,Z), append(Z,[X],Y).

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

We assume that the code has been analyzed to determine types, etc. The following code for append is a specialized version of the predicate. It is compiled into the following extended WAM code.

```
nrev/2: /* code for nrev */

append/3:
        await_nonvar 0              % wait until X0 ground
                                    % or leftmost
        switch_on_term fail La Ll fail

La:     get_nil X0    % first arg is nil
        get_value X1 X2             % X = X
        proceed

Ll:     get_list X0                 % first arg is list
        unify_variable X3           % X3 <- X
        unify_variable X0           % X0 <- Xs
        lock_and_get_list X2 X4     % lock X2, list in X4
        unify_x_value X3            % second arg of list: X
        unify_x_variable X5         % X5 <- Zs
        unlock X2 X4                % unlock X2, X2=[X|Zs]
        put_value X5 X2             % X2 <- Zs
        execute append/3
```

`lock_and_get_structure`(F, i, n)
: If $\mathtt{X}_i$ contains a variable, then write mode is entered and a structure with the functor F is initiated on the heap. A pointer to the new structure is stored in $\mathtt{X}_n$. If $\mathtt{X}_i$ contains a structure and the functor of the structure is F, the `S` register is set to point to the first argument of the structure, otherwise failure occurs.

`lock_and_get_list`(i, n)
: If $\mathtt{X}_i$ contains a variable, then write mode is entered and a list is initiated on the heap. A pointer to the new list is stored in $\mathtt{X}_n$. If $\mathtt{X}_i$ contains a list, the

S register is set to point to the first argument of the list, otherwise failure occurs.

`unlock(`*i*, *n*`)`
: This instruction is ignored in read mode. In write mode the contents of $\mathtt{X}_i$ is unified with the contents of $\mathtt{X}_n$.

Runtime Test Instructions

These instructions are used for doing runtime tests in order to enforce global binding-determinism. The following program illustrates the need for synchronization. Each recursion level must wait until its N variable becomes instantiated before calculating the addition. In practice, this program is compiled as a call to a built-in reduction predicate, as described above.

```
sum([],S,S).
sum([X|Xs],N,S) :- N2 is X + N, sum(Xs,N2,S).
```

It is compiled into the following extended WAM code.

```
sum/3:  switch_on_term Lv La Ll fail
Lv:     try La
        trust L1

La:     get_nil_x0
        get_value X1 X2
        proceed

L1:     get_list X0
        unify_variable X4
        unify_variable X0
        builtin '$plus' X1 X1 X4
        execute sum/3

Ll:     build_rec_poslist X0 X3 X4 X0 % create vector list Xs
        build_variables X1 X3 X5 X1   % create vector list NN2
        start_left_body L2            % start parallel phase
        execute sum/3                 % execute base case

L2:     initialize_left 1             % initialize worker
L3:     spawn_left 1 X2 G3            % while X2 < G3 do
        put_nth_head G4 X2 0 X0       % X <- Xs[X2+0]
        put_nth_head G5 X2 0 X1       % N <- NN2[X2+0]
        put_nth_head G5 X2 1 X3       % N2 <- NN2[X2+1]
        await_strictly_nonvar X1      % wait until N bound
        builtin '$plus' X4 X0 X1      % X4 <- N + X
```

```
        get_value X4 X3              % unify N X4
        jump L3                      % next rec. level
```

`await_leftmost`
: This instruction forces the current recursion level to suspend until it is leftmost in the resolvent, i.e., all preceding recursion levels have terminated.

`await_nonvar(`i`)`
: This instruction forces the current recursion level to suspend until either the dereferenced contents of $\texttt{X}_i$ is a non-variable, or the current recursion level is leftmost in the resolvent, i.e., all preceding recursion levels have terminated.

`await_strictly_nonvar(`i`)`
: This instruction forces the current recursion level to suspend until the dereferenced contents of $\texttt{X}_i$ is a non-variable. If the recursion level becomes leftmost in the resolvent and the result of dereferencing $\texttt{X}_i$ is an unbound variable, then a run time error is signalled.

`await_variable(`i`)`
: This instruction forces the current recursion level to suspend until it is leftmost in the resolvent. If the variable referred to by $\texttt{X}_i$ becomes bound during suspension, a run time error is signalled.

5.6 EXECUTION EXAMPLE

Suppose we have the program for naive reverse given on page 42 and the goal `nrev([a,b,c,d],Z1)`. Assume that the program is executed using 4 workers. Before the parallel phase is initiated two vector lists are constructed on the heap (`[a,b,c,d]` and `[Z1,Z2,Z3,Z4,Z5]`), and a call to the base case is executed (`nrev([],Z5)`) with the result that `Z5` is bound to `[]`. Static scheduling and horizontal mapping are used to distribute the recursion levels in the following way.

Worker	Recursion level
1	`append(Z5,[d],Z4)`
2	`append(Z4,[c],Z3)`
3	`append(Z3,[b],Z2)`
4	`append(Z2,[a],Z1)`

Figure 5.5 shows a snapshot of the parallel execution. When the parallel phase begins only worker 1 is active. As soon as `Z4` becomes bound worker 2 is activated, as soon as `Z3` becomes bound worker 3 is activated, an so on. Because workers with lower numbers are ahead of workers with higher numbers there will usually be very little suspension once all workers have started.

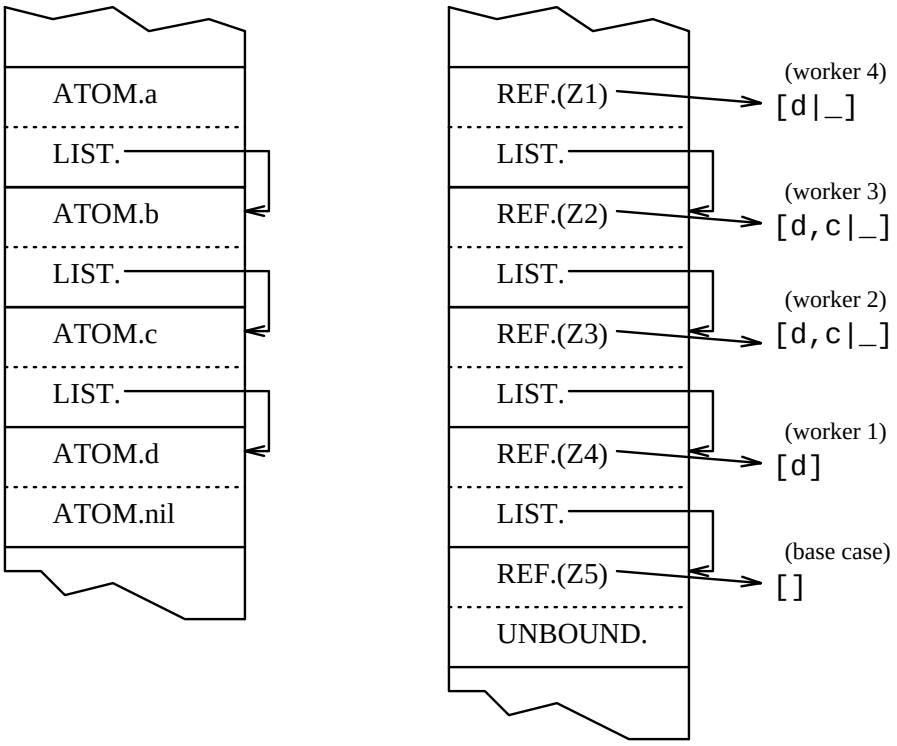


Figure 5.5: Snapshot of parallel execution of `nrev([a,b,c,d],Z)` using 4 workers. Worker 3 and 4 are waiting for worker 2 to bind the tail of the list `Z3` to `[]`.

EVALUATION

IN this chapter we evaluate the proposed machine. We present the results obtained when running several benchmarks in Reform Prolog on a parallel machine and compare them with the results obtained when running the same program in the unextended sequential Prolog implementation. Potential bottlenecks in the Reform engine are discussed. We also compare the efficiency of Reform Prolog with other parallel Prolog systems—AND-in-OR parallel, stream-AND-parallel, and dependent AND-parallel systems. We have not been able to compare with independent AND-parallel implementations.

6.1 METHODOLOGY

Reform Prolog has been implemented on two machines: Sun 630 MP and Sequent Symmetry. Both these machines are bus-based, cache-coherent shared-memory architectures. The Sun has 4 Sparc processors and the Sequent has 26 Intel 80386 processors. The experiments described in this chapter were conducted on the Sequent Symmetry. In our experiments we used 24 of the available 26 processors on the Symmetry, the remaining two were left for operating system activities.

The metric we use for evaluating parallelization is the speed-up it yields. We present *relative* and *normalized* speed-ups. Relative speed-up expresses the ratio of execution time of the program (compiled with parallelization) on a single processor to the execution time using p processors. Normalized speed-up expresses the ratio of execution time of the original sequential program (compiled without any extensions for parallelism) on a single processor to the execution time using p processors.

6.2 BENCHMARKS

We have parallelized four benchmark programs. Two programs (Match and Tsp) are considerably larger than the other two. One program (Map) exploits in-

dependent AND-parallelism, whereas the other three exploit dependent AND-parallelism. The execution times are measured in seconds walltime.

Map. The map program maps a function on each element of a list, producing a new list. The function used in the benchmark does nothing useful, it only decrements a counter 100 times. A list of 10000 elements was used.

Workers	Execution time (s)	Relative speed-up	Normalized speed-up
1	40.40	1.00	0.98
4	10.12	3.99	3.89
8	5.07	7.96	7.76
16	2.54	15.91	15.50
24	1.70	23.76	23.15

Sequential time: 39.59 sec.

An interesting question is whether this speed-up is maintained when the execution time of each recursion level is made smaller. It turned out that the same speed-up is achieved for recursion levels 1/4 the size of the one measured above. If each recursion level is made smaller than that, the sequential part begins to dominate. The parallel part of the computation continued to show the same speed-up, however. We therefore executed the same program using integer recursion over an array (not taking the time to construct the array into account). That program showed linear speed-up for all sizes of the recursion level (even with an empty body).

Nrev. The nrev program reverses a list using list concatenation ('naive reverse'). A list of 900 elements was used. This length of the list was chosen because it gave a measurable execution time.

Workers	Execution time (s)	Relative speed-up	Normalized speed-up
1	30.80	1.00	0.88
4	8.08	3.81	3.43
8	3.96	7.77	6.99
16	2.01	15.32	13.78
24	1.36	22.65	20.36

Sequential time: 27.70 sec.

We believe that a large portion of the parallelization overhead, for this program (12%), originates from the instruction decoding overhead. This belief is based on

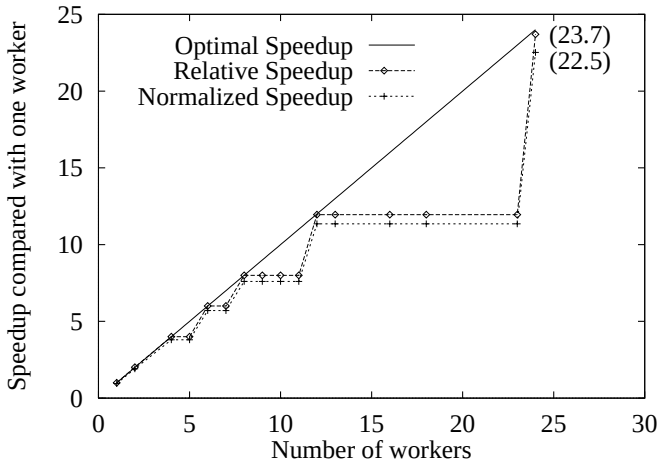


Figure 6.1: Plot of speed-up of the match benchmark. The plot illustrates the effect of executing a small number of recursion levels compared with the set of workers.

the fact that the parallel code for append/3 contains about 35% more instructions than the sequential program. In a native code implementation the parallelization overhead would be considerably less, possibly in the range of 3%.

Match. The match program implements a dynamic programming algorithm for comparing, e.g., DNA-sequences. We use a variant of the algorithm described by Smith and Waterman [43]. The result of each single comparison is inserted in an incomplete sorted binary tree. One sequence of length 32 was compared with 24 other sequences.

Workers	Execution time (s)	Relative speed-up	Normalized speed-up
1	68.88	1.00	0.95
4	17.22	3.99	3.80
8	8.61	7.99	7.60
16	5.76	11.95	11.35
24	2.91	23.70	22.52

Sequential time: 65.44 sec.

If 24 recursion levels are divided between 16 workers then 8 workers are going to execute two recursion levels each and 8 are going to execute one recursion level each. The plot in Figure 6.1 illustrates the effect of this.

Tsp. The tsp program implements an approximation algorithm for the Travelling Salesman problem, as described by Tucker [48]. A tour of 45 cities was computed.

Workers	Execution time (s)	Relative speed-up	Normalized speed-up
1	258.22	1.00	0.90
4	68.85	3.75	3.37
8	34.55	7.47	6.73
16	17.25	14.96	13.47
24	11.82	21.85	19.67

Sequential time: 232.40 sec.

6.3 DISCUSSION

There are several aspects of parallel computation which are interesting, apart from speed-up. In order to anticipate how well other programs will behave in our system, one must also take into account load balance, parallelization overhead, etc. We present this information for the benchmarks we have executed, and discuss its relevance for other programs.

Load Balance

One way of estimating the load balance of a computation is to compare the finishing time of the workers. We measured the execution time for each worker when executing our benchmarks. Static scheduling was used in all experiments.

Map. This program displayed a very uniform load balance (less than 0.3% difference between workers). This is hardly surprising since the number of recursion levels executed by each worker is large, and there is no difference in execution time between recursion levels.

Nrev. The execution time of each worker only varied about 3% when executing this program. There is a slight difference in the execution time of each recursion level, but the large number of recursion levels executed by each worker smooths the differences.

Match. When 24 workers were used the execution time varied less than 0.3% between workers. This is explained by the fact that each worker executed one recursion level each, and that all recursion levels executed in the same time.

Tsp. This program displayed an uniform load balance on all but three workers. This is explained by the fact that 45 recursion levels were executed in all; 21 workers executed 2 recursion levels each while 3 workers executed 1 recursion level each. Despite this situation, tsp displays good speed-up (21.85). Using dynamic scheduling would not have improved the results in this case.

It is not surprising that programs containing data dependencies appear to be well balanced, using this way of measuring, since the data dependencies force fast recursion levels to wait for slower ones. We therefore measured the total suspension time for each worker. The results showed that no worker was suspended more than 0.6% of the total execution time when using 24 processors.

Sequential Fraction of Execution Time

The sequential part of a program determines its maximal speed-up, according to Amdahl's law [3]. We therefore measured the sequential parts of each of our benchmark programs. The sequential part includes everything except the parallel execution—traversing the recursive arguments as well as setting up the global environment for the parallel phase and executing the sequential Prolog code. The following table shows for each program which fraction of the total execution time is not subject to parallelization.

Program	Sequential fraction
Map	0.3 %
Nrev	0.04 %
String	0.003 %
Tsp	0.005 %

We conclude from these data that the unparallelized parts represent negligible fractions of the total execution times. Another conclusion is that there is no point in parallelizing the head unification of parallelized predicates, since it represents such a tiny fraction of the computation. If programs that are much more fine-grained are to be parallelized, then this conclusion might no longer hold. It cannot be concluded that, e.g., the nrev benchmark, can be speeded up 2500 times—the maximal speedup is 900 since 900 recursion levels are execute.

Parallelization Overhead

We have calculated the parallelization overhead of Reform Prolog from the figures in the preceding section, and estimated the parallelization overhead one might expect in a native code implementation. This was done by removing the overhead for decoding a larger number of machine instructions from the parallel execution

time. The actual parallelization overhead and the estimated overhead for native code (using the above method) is shown in the table below.

Program	Emulated Overhead	Expected Overhead
Map	2 %	2 %
Nrev	12 %	3 %
Match	5 %	4 %
Tsp	10 %	7 %

Our hypothesis is that this overhead originates mostly from the more complicated dereferencing and binding operations.

Parallel Efficiency

Let us define

$$\text{relative efficiency} = \frac{\text{relative speed-up on } N \text{ processors}}{N}$$

$$\text{normalized efficiency} = \frac{\text{normalized speed-up on } N \text{ processors}}{N}$$

We then obtain the following results for our experiments when executing each program on 24 processors.

Program	Relative efficiency	Normalized efficiency
Map	99 %	96 %
Nrev	95 %	83 %
Match	99 %	94 %
Tsp	91 %	82 %

6.4 BOTTLENECKS AND POTENTIAL PROBLEMS

The sequential part of a recursion parallel computation is dominated by the time it takes to build the vector lists. For coarse-grained programs the sequential part of the computation remains insignificant, but for fine-grained programs it starts to dominate. A possible solution is to parallelize this part, e.g., by using methods for parallel unification similar to those presented by Barklund [5].

Another potential bottleneck is the use of operating system semaphores. The semaphores supplied by the operation system are slow, activating a suspended worker using semaphores might take as long as 5 milliseconds on a Sun 630 MP. Activating suspended workers using user-implemented spin-lock semaphores require less than one millisecond on the same machine. The advantage of using

operating system supplied semaphores is that other processes are given access to the machine when there is no active recursion parallel computation.

A potential problem is that vector lists (containing variables) are used to communicate bindings between recursion levels, e.g., the second argument of `sum/3` (see page 33). This method for passing information from one recursion level to another is consuming considerable amounts of heap space, heap space not consumed in sequential versions of the program. We are considering using bounded buffers instead. This is possible since the heap space used to represent a variable can be reused once the binding of the variable has been communicated. This would result in heap space consumption (for this purpose) proportional to the number of workers instead of proportional to the number of recursion levels to be executed.

6.5 OTHER SYSTEMS

We compare Reform Prolog with the other Prolog systems which supports dependent AND-parallelism, that we are aware of: Andorra-I [52], Shen's prototype implementation of DASWAM [41], and NUA-Prolog [37]. All three are compiler-based implementations using abstract machines similar to WAM. Naish's PNU-Prolog [36] also supports dependent AND-parallelism but we were not able to find any published performance figures for that system.

It should be noted that these systems to some extent exploit different forms of parallelism. Reform Prolog and NUA-Prolog exploit deterministic dependent AND-parallelism (recursion parallelism in the case of Reform Prolog). Andorra-I exploits deterministic dependent AND-parallelism and OR-parallelism (here we are only interested in the AND-parallel component of the system). DASWAM exploints unrestricted dependent AND-parallelism.

Unfortunately, we can only make a very limited comparison with NUA-Prolog, since the published benchmark programs stress the constraint-solving capabilites of the system, rather than its potential for raw AND-parallel speed-up. However, we have compared their result on the nrev benchmark with ours.

In addition to comparing Reform Prolog with AND-parallel systems we have also made a limited comparison with JAM [13] (an implementation of Parlog which exploit stream AND-parallelism) and with Muse [2, 20] (an OR-parallel implementation of Prolog). When comparing with Muse we executed the map benchmark. This was done using the findall technique for AND-in-OR parallelism described by Carlsson, Danhof and Overbeek [11].

Parallelization Overhead

The Andorra-I system shows parallelization overheads of an average of 40% on a set of benchmarks [52] compared with the sequential version of Andorra-I. NUA-Prolog

shows a parallelization overhead of 55% on the nrev benchmark. The prototype implementation of DASWAM exhibits a parallelization overhead of more than 100%. The parallelization overhead of Muse, when executing the map program using AND-in-OR, was in the range of 3%. JAM is reported to have an overhead of 100% compared with sequential Prolog.

The parallelization overhead of Reform Prolog is 2–12%. This is considerably less than any of the other systems, with the exception of Muse which obtained its low figure for independent AND-parallelism. We believe that the reasons for the low overhead for Reform Prolog are the following.

1. In Reform Prolog a large part of the parallelization work is done at compile time. This enables the use of a simple abstract machine.

2. Reform Prolog exploits a limited amount of the available parallelism. This simplifies scheduling and process management.

Parallel Efficiency

We have calculated the parallel efficiency from published figures. In the case of Andorra-I [52] we have excluded benchmarks that mainly exhibit OR-parallelism. The Andorra-I figures were obtained on a Sequent Symmetry using 10 processors. The NUA-prolog [37] figure was obtained from the execution of the nrev benchmark using 11 processors on a Sequent Symmetry. The Muse figure was obtained from executing the map benchmark using 10 processors on a Sequent Symmetry. We were not able to find any figures for JAM and DASWAM. The Reform figures were obtained using 24 processors on a Sequent Symmetry as described above (see Section 6.3) (this is more than twice the number of processors the other systems used—using less processors improves the result).

System	Relative efficiency	Normalized efficiency
Andorra-I	33–94%	26–66%
NUA-Prolog	71%	36%
Muse	33%	32%
Reform Prolog	91–99%	82–96%

This comparison can only be viewed as a rough estimate of the actual differences between the systems. In order to make a better comparison one would have to execute similar programs on all systems and compare the obtained figures. Unfortunately, this has not been possible for practical reasons.

We nevertheless conclude that the efficiency of Reform Prolog is high by comparison. The reasons underlying this are both low parallelization overhead and good

speed-up on all executed benchmark programs. We believe the good speed-up to be due to efficient process management and scheduling. The fact that all processes can be initiated simultaneously is an important advantage over other systems. This in turn is an effect of using Reform Compilation for data-parallelism, as a method for parallelizing Prolog.

DISCUSSION

7.1 RELATED WORK

There are two sources of parallelism that can be exploited when parallelizing Prolog: OR-parallelism and AND-parallelism. Aurora [26] and Muse [2] are the two outstanding implementations that exploit OR-parallelism. AND-parallel implementations can be divided into several categories: independent AND-parallel and dependent AND-parallel. Several methods have been proposed to exploit independent AND-parallelism, the model proposed by DeGroot [15] being the first, Hermenegildo [17] and Lin and Kumar [23] have also developed schemes for independent AND-parallel execution. Restricted dependent AND-parallelism have been exploited by several logic programming implementations, e.g., Naish's [36] parallel NU-Prolog. The dependent AND-parallelism is restricted in such a way that only binding deterministic goals are executed in parallel. Shen [40] recently proposed a scheme for full dependent AND-parallel execution of Prolog.

There are also several combinations of OR-parallelism and AND-parallelism, e.g., the Andorra-I [39] system, developed at Bristol University, which exploits restricted dependent AND-parallelism.

Reform Prolog exploits restricted dependent AND-parallelism using a method which is most closely related to Naish's PNU-Prolog.

Bounded quantifications is another way to exploit data-parallelism in Prolog. Barklund and Millroth [6] and Voronkov [50], among others, have discussed how bounded quantifications can be used in Prolog. Preliminary results indicate that bounded quantifications can be executed in parallel on shared-memory multiprocessors using the abstract machine proposed in this thesis. The implementation of WAM, on which Reform Prolog is based, has also been extended for parallel execution of bounded quantifications on a SIMD (single instruction stream, multiple data stream) computer, Thinking Machines Corp. model CM-2. Arro, Barklund and Bevemyr [4] report promising preliminary results from this implementation.

7.2 FUTURE WORK

In the continuation of this work we propose to further investigate scheduling properties, especially for fine grained programs and for programs with nonuniform load balance. It is possible that existing scheduling algorithms, developed for loop programs without data-dependencies, are inadequate for efficient scheduling of recursion parallelism. For executing large programs, a parallel garbage collector has to be designed.

It is possible that parallel unification is needed to execute fine-grained programs. This is something we propose to investigate. Lindgren will develop a native code compiler for Reform Prolog. The implementation techniques presented in this thesis might need modifications to be used in a native code setting.

Another interesting area of research is to find ways to combine recursion parallelism with other forms of parallelism, e.g, OR-parallelism. It is possible that nested recursion parallelism has to be considered in order to achieve speed-up for some programs; possibly, we also have to extend the set of programs that can be parallelized. This will have to be investigated.

7.3 CONCLUSION

We have designed and implemented an efficient execution model for parallel execution of Prolog, based on the data-parallel approach. We have implemented the execution model through extensions to WAM. The resulting machine has been shown to utilize a MIMD parallel computer efficiently and with a low overhead for parallelization. One of the main reasons behind this is that the goal of the execution model is to exploit enough parallelism to make efficient use of the parallel computer, whereas many other parallel implementations of Prolog have tried to exploit as much parallelism as possible. We believe that to be a serious mistake.

The other significant advantage of the execution model is that it is easy to realize which parts of a program are going to execute in parallel—an effect of data-parallelism. This makes it easy to write efficient programs—one only has to write deterministic programs that perform list or integer recursion.

Tick listed the main implementation problems and their possible solutions for the existing parallel implementations in an advanced tutorial on parallel logic programming [45].

1. Memory consumption is excessive because: 1) single assignment, 2) no backtracking in concurrent languages, 3) streams implemented as list in concurrent languages.

Solutions: 1) new garbage collection algorithms, 2) approximative reference counting, 3) static reference counting, 4) language de-evolution, 5) multi-lingual programming.

2. Task granularity is too fine, giving abundant supply of small concurrent goals.

   Solutions: 1) static granularity estimation, 2) program sequentialization (by user or compiler), 3) explicit distribution pragma.

3. Inability to exploit stack-based execution and all its benefits.

   Solutions: 1) clever memory management, 2) sequentialization.

4. Guaranteeing fairness of task reduction is difficult.

   Solutions: 1) goal priorities, 2) time-slicing.

Reform Prolog suffers from the first problem and to some degree from the third. It uses single assignment (in the same way as all WAM based implementations). A more serious problem is that Reform Prolog is unable to exploit stack-based execution in some situations—vectors on the heap are used instead of environments on the stack. We intend to ameliorate this problem by using a fast and efficient garbage collection algorithm. Timestamping variables enables the use of a copying garbage collector. Such a garbage collector should be linearly time dependent on the size of the *live* data, instead of the size of the memory area to garbage collect.

Bibliography

1. Aït-Kaci, H., *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, Mass., 1991. {4}

2. Ali, K. A. M. and Karlsson, R., The Muse Approach to OR-parallel Prolog, in *International Journal of Parallel Programming*, 19:129–162, 1990. {53, 57}

3. Amdahl, G. M., Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities, in *AFIPS Conference Proceedings*, American Federation of Information Processing Societies, 1967. {51}

4. Arro, H., Barklund, J. and Bevemyr, J., Parallel Bounded Quantifications – Preliminary Results, UPMAIL Technical Report No. 74, Computing Science Department, Uppsala University, 1993. {57}

5. Barklund, J., *Parallel Unification*, Ph.D. Thesis, Computing Science Department, Uppsala University, 1990. {52}

6. Barklund, J. and Millroth, M., Providing Iteration and Concurrency in Logic Programs through Bounded Quantifications, in *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, Ohmsha, Tokyo, 1992. {57}

7. Bevemyr, J., The Luther WAM Emulator, UPMAIL Technical Report No. 72, Computing Science Department, Uppsala University, 1992. {4}

8. Carlsson, M., SICStus Prolog Internals Manual, Internal Report, Swedish Institute of Computer Science, 1989. {4}

9. Carlsson, M., *Design and Implementation of an OR-Parallel Prolog Engine*, Ph.D. Thesis, Swedish Institute of Computer Science, 1990. {23}

10. Carlsson, M., Freeze, Indexing, and Other Implementation Issues in the WAM, in *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, Cambridge, Mass., 1987. {24}

11. Carlsson, M., Danhof, K. and Overbeek, R., A Simplified Approach to the Implementation of AND-Parallelism in an OR-Parallel Environment, in *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass., 1988. {53}

12. Clocksin, W. F., Design and Simulation of a Sequential Prolog Machine, in *New Generation Computing*, 3:101–120, 1985. {20}

13. Crammond, J., The Abstract Machine and Implementation of Parallel Parlog, in *New Generation Computing*, 10:385–422, 1992. {53}

14. Crammond, J., Scheduling and Variable Assignment in the Parallel Parlog Implementation, in *Proceedings of the North American Conference on Logic Programming*, Austin, MIT Press, Cambridge, Mass., 1990. {17, 27}

15. DeGroot, D., Restricted AND-parallelism, in *Proceedings of the International Conference on Fifth Generation Computer Systems*, Institute for New Generation Computing, 1984. {57}

16. Hatcher, P. J. and Quinn, M. J., *Data-parallel Programming on MIMD Computers*, MIT Press, Cambridge, Mass., 1991. {30}

17. Hermenegildo, M. V., An Abstract Machine for Restricted AND-parallel Execution of Logic Programs, in *Third International Conference on Logic Programming*, Springer-Verlag, Berlin, 1986. {57}

18. Hermenegildo, M. V., Relating Goal Scheduling, Precedence, and Memory Management in AND-parallel Execution of Logic Programs, in *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, Cambridge, Mass., 1987. {27}

19. Karlsloot, M. and Tick, E., Sequentializing Parallel Programs, in *Proceedings of the Phoenix Seminar and Workshop on Declarative Programming*, Springer-Verlag, Berlin, 1991. {23}

20. Karlsson, R., *A High Performance OR-parallel Prolog System*, Ph.D. Thesis, Swedish Institute of Computer Science, 1992. {23, 53}

21. King, A. and Soper, P., Schedule Analysis of Concurrent Logic Programs, in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, Cambridge, Mass., 1992. {23}

22. Kruskal, C. P. and Weiss, A., Allocating Independent Subtasks on Parallel Processors, in *IEEE Transactions on Software Engineering*, 11:1001–1016, 1991. {27}

23. Lin Y.-J. and Kumar, V., AND-parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results, in *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass., 1988. {57}

24. Lindgren, T., *The Compilation and Execution of Recursion Parallel Logic Programs for Shared Memory Multiprocessors*, Ph.L. Thesis, Computing Science Department, Uppsala University, 1993. {14, 32, 35, 36}

25. Lindholm, T. and O'Keefe, R. A., Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code, in *Logic Programming: Proceedings of the Fourth International Conference*, MIT Press, 1987. {30}

26. Lusk, E., Warren, D. H. D. and Haridi, S., The Aurora OR-parallel Prolog System, in *New Generation Computing*, 7:243–271, 1990. {57}

27. Mannila, H. and Ukkonen, E., Timestamped Term Representation for Implementing Prolog, in *Proceedings 1986 Symposium on Logic Programmin*, IEEE Computer Society Press, Washington, D.C., 1986. {20}

28. Mariën, A. and Demoen, B., A new Scheme for Unification in WAM, in *Logic Programming: Proceedings of the 1991 International Symposium*, MIT Press, Cambridge, Mass., 1991. {18}

29. Markatos, E. P. and LeBlanc, T. J., Using Processor Affinity in Loop Scheduling on Shared-Memory Multiprocessors, Technical Report 410, University of Rochester, 1992. {27}

30. Mellor-Crummey, J. M. and Scott, M. L., Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors, in *ACM Transactions on Computer Systems*, 9:21–65, 1991. {16}

31. Millroth, H., *Reforming Compilation of Logic Programs*, Ph.D. Thesis, Computing Science Department, Uppsala University, 1990. {3, 9, 11}

32. Millroth, H., Compiling List Processing in Parallelized Prolog, unpublished manuscript, 1992. {34}

33. Millroth, H., Reforming Compilation for Nonlinear Recursion, in *Proceedings of the International Conference on Logic Programming & Automated Reasoning*, LNCS 624, Springer-Verlag, Berlin, 1992. {12}

34. Millroth, H., SLDR-Resolution: Parallelizing Structural Recursion in Logic Programs, to appear in *Massively Parallel Reasoning Systems*, editors Robinson, J. A. and Sibert, E. E., MIT Press, Cambridge, Mass. {11}

35. Moto-oka, T. *et al.*, Challenge for Knowledge Information Processing Systems, in *Proceedings of International Conference on Fifth Generation Computer Systems*, 1981. {2}

36. Naish, L., Parallelizing NU-Prolog, in *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, MIT Press, Cambridge, Mass., 1988. {11, 17, 18, 53, 57}

37. Palmer, D. and Naish, L., NUA-Prolog: An Extension to the WAM for Parallel Andorra, in *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, Cambridge, Mass., 1991. {53, 54}

38. Polychronopoulos, C. D. and Kuck, D. J., Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers, in *IEEE Transactions on Computers*, C-36(12), 1987. {27}

39. Santos Costa, V., Warren, D. H. D. and Yang, R., The Andorra-I Engine: A Parallel Implementation of the Basic Andorra Model, in *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, Cambridge, Mass., 1991. {57}

40. Shen, K., Exploiting Dependent And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS), in *Proceedings of the Joint International Conference and Symposium on Logic Programming*, MIT Press, Cambridge, Mass., 1992. {12, 57}

41. Shen, K., *Studies of AND/OR Parallelism in Prolog*, Ph.D. Thesis, University of Cambridge, 1992. {18, 20, 53}

42. Singh, J. P. and Hennessy, J. L., An Empirical Investigation of the Effectiveness and Limitations of Automatic Parallelization, in *Proceedings of the International Symposium on Shared Memory Multiprocessing*, 1991. {1}

43. Smith, T. F. and Waterman, M. S., Identification of common molecular subsequences, *Journal of Molecular Biology*, 147:195–196, 1981. {49}

44. Tang, P. and Yew, P.-C., Processor Self-Scheduling for Multiple Nested Parallel Loops, in *Proceedings 1986 International Conference on Parallel Processing*, 1986. {26}

45. Tick, E., An Advanced Tutorial in Parallel Logic Programming, tutorial held at the Joint International Conference and Symposium on Logic Programming, Washington, 1992. {58}

46. Tick, E., *Parallel Logic Programming*, MIT Press, Cambridge, Mass., 1991. {2}

47. Touati, H. and Despain, A., An Empirical Study of the Warren Abstract Machine, in *Proceedings 1987 Symposium on Logic Programming*, IEEE Computer Society Press, 1987. {20}

48. Tucker, A., *Applied Combinatorics*, Second Edition, pp. 97–104, John Wiley & Sons, Chichester, 1984. {50}

49. Tärnlund, S.-Å., Reform, unpublished manuscript, Computing Science Department, Uppsala University, 1991. {9}

50. Voronkov, A., Logic Programming with Bounded Quantifiers, in *Logic Programming—Proceedings of the Second Russian Conference on Logic Programming*, Springer-Verlag, Berlin, 1992. {57}

51. Warren, D. H. D., An Abstract Prolog Instruction Set, SRI Tech. Note 309, SRI International, Menlo Park, Calif., USA, 1983. {4}

52. Yang, R., Beaumont, T., Dutra, I., Santos Costa V. and Warren D. H. D., Performance of the Compiler-based Andorra-I System, submitted for publication, Department of Computer Science, University of Bristol. {53, 54}

Benchmarks

A.1 MAP

```
test_map(N, L, Time) :-
    random_list(N, L),
    statistics(walltime, [T1|_]),
    map(L, R),
    statistics(walltime, [T2|_]),
    Time is T2-T1.

:- parallel map/2.

map([],[]).
map([X|Xs],[Y|Ys]) :- rel(X,Y), map(Xs,Ys).

rel(X, 1) :- count(100).

count(0) :- !.
count(N) :- N2 is N - 1, count(N).
```

A.2 NAIVE REVERSE (NREV)

```
test_nr(N, R, Time) :-
    random_list(N, L),
    statistics(walltime, [T1|_]),
    nrev(L, R),
    statistics(walltime, [T2|_]),
    Time is T2-T1.

:- parallel nrev/2.

nrev([],[]).
nrev([X|Xs], Y) :- nrev(Xs,Z), append(Z,[X],Y).

append([],X,X).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

A.3 COMPARING DNA SEQUENCES (MATCH)

```
% Sequence matching (sequences has length 32 in the test example)
% Matches N random sequences against a random target sequence.

:- ensure_loaded(utilities).

% test_seq(+N, -Res, -Time):

test_seq(N, Res, Time) :-
    random_sequence(32, Side),
    random_sequences(N, 32, Tops),
    statistics(walltime, [T1|_]),
    match_sequences(Tops, Side, Res),
    statistics(walltime, [T2|_]),
    Time is T2-T1.

/************************************************/

:- parallel match_sequences/3.

match_sequences([], _, []).
match_sequences([Top|Tops], Side, Tree) :-
    match_two_seq(Side, Top, none, entry(0,0,0,0), Sim),
    put_in_tree(Tree, Sim),
    match_sequences(Tops, Side, Tree).

match_two_seq([], _, _, Res, Res).
match_two_seq([S|Side], Top, Prev, _, Res) :-
    match(Top, S, Prev, Row, entry(0,0,0,0), entry(0,0,0,0), L),
    match_two_seq(Side, Top, Row, L, Res).

match([], _, _, [], _, L, L).
match([T|Top], S, none, [E|Row], Zero, W, L) :- !,
    match_entry(T, S, Zero, Zero, W, E),
    match(Top, S, none, Row, Zero, E, L).
match([T|Top], S, [N|Prev], [E|Row], NW, W, L) :-
    match_entry(T, S, NW, N, W, E),
    match(Top, S, Prev, Row, N, E, L).

match_entry(T, S, Entry1, Entry2, Entry3, Entry4) :-
    Entry1 = entry(NWmax,NWnw,_,_),
    Entry2 = entry(Nmax,Nnw,Nn,_),
    Entry3 = entry(Wmax,Wnw,_,Ww),
    Entry4 = entry(Emax,Enw,En,Ew),
    alpha_beta_penalty(Wnw, Ww, Ew),
    alpha_beta_penalty(Nnw,Nn,En),
    match_weights(T, S, Weight),
    Enw1 is NWnw + Weight,
```

```
    maxl([0,Ew,En], Enw1, Enw),
    maxl([NWmax,Wmax,Nmax], Enw, Emax).

alpha_beta_penalty(X, Y, Z) :-
    X1 is X - 4,            % alpha (indel) penalty
    Y1 is Y - 1,            % beta (extension) penalty
    max(Y1, X1, Z).

maxl([], X, X).
maxl([Y|R], X, Z) :-
    max(X, Y, X1),
    maxl(R, X1, Z).

max(X, Y, Z) :- X<Y, !, Z = Y.
max(X, Y, X).

% Dummy weights

match_weights(X, X, 1) :- !.
match_weights(_,_,0).

put_in_tree(X,Y) :-
    var(X), !,
    X = t(_, Y, _).
put_in_tree(t(_, X, _), Y) :-
    X == Y, !.
put_in_tree(t(L, Y, _R), X) :-
    X @< Y, !,
    put_in_tree(L, X).
put_in_tree(t(_L, _Y, R), X) :-
    put_in_tree(R, X).

/************************************************/

random_sequence(0, []) :- !.
random_sequence(N, [X|Xs]) :-
    random(10, X),
    N1 is N-1,
    random_sequence(N1, Xs).

random_sequences(0, _, []) :- !.
random_sequences(N, I, [X|Xs]) :-
    random_sequence(I, X),
    N1 is N-1,
    random_sequences(N1, I, Xs).
```

A.4 TRAVELING SALESMAN (TSP)

```
% Near-optimal solution to the Traveling Salesman Problem.

% test_tsp(+N, -Path, -Cost, -Time):

test_tsp(N, Path, Cost, Time) :-
    random_matrix(N, Matrix),
    statistics(walltime, [T1|_]),
    tsp(N, N, Matrix, Path, Cost),
    statistics(walltime, [T2|_]),
    Time is T2-T1.

/************************************************/

% tsp(+N, -Path, -Cost):
% Solves the TSP for a random graph of N nodes.

tsp(N, Path, Cost) :-
    random_matrix(N, Matrix),
    tsp(N, N, Matrix, Path, Cost).

tsp(V, N, Matrix, Path, Cost) :-
    tsp(V, N, Matrix, _, 100000, Path, Cost).

:- parallel tsp/7.

tsp(0, _, _, P, C, P, C) :- !.
tsp(V, N, Matrix, MinP, MinC, P, C) :-
    all_but_this_vertex(N, V, Vs1),
    travel(Vs1, Matrix, [V], P1),
    cost([V|P1], Matrix, C1),
    update_minimum1(C1, P1, V, MinC, MinP, MinC1, MinP1),
    V1 is V-1,
    tsp(V1, N, Matrix, MinP1, MinC1, P, C).

all_but_this_vertex(0, _, []) :- !.
all_but_this_vertex(N, N, Vs) :-
    !,
    N1 is N-1,
    all_but_this_vertex(N1, N, Vs).
all_but_this_vertex(N, V, [N|Vs]) :-
    N1 is N-1,
    all_but_this_vertex(N1, V, Vs).

travel([], _, P, P).
travel([V|Vs], Matrix, P0, P) :-
    minimal_cost([V|Vs], Matrix, P0, U, W),
    delete_vertex([V|Vs], U, Vs1),
```

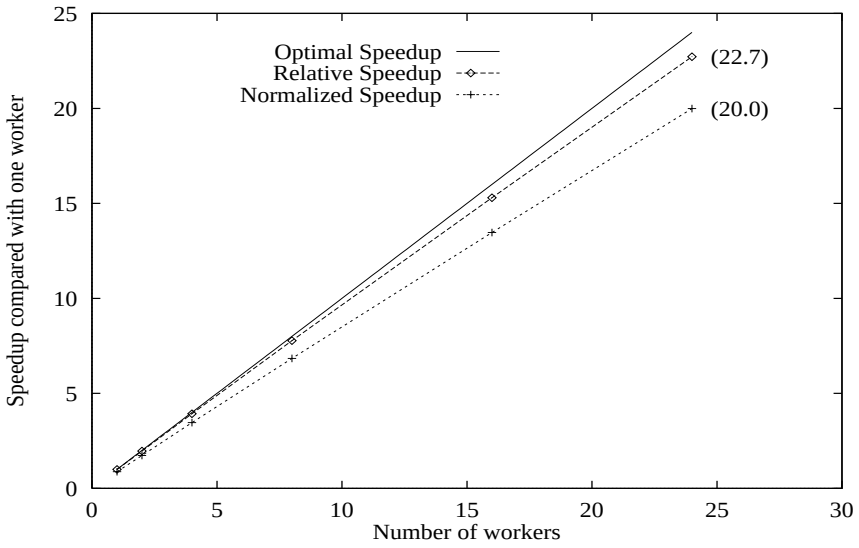
```
    append(Prefix, [W|Suffix], P0),
    append(Prefix, [U,W|Suffix], P1),
    travel(Vs1, Matrix, P1, P).

% minimal_cost(Verteces+, Matrix+, Path+, U-, W-):
% U is a vertex among Verteces and W a vertex in Path, such that
% the distance from U to W is minimal.

minimal_cost(Verteces, Matrix, Path, U, W) :-
    minimal_cost(Verteces, Matrix, Path, U, W, 100000, _, _).

minimal_cost([], _, _, U, W, _, U, W).
minimal_cost([V|Vs], Matrix, Path, U, W, MinC, MinU, MinW) :-
    arg(V, Matrix, Row),
    row_min(Path, Row, W1, C),
    update_minimum2(C, V, W1, MinC, MinU, MinW, MinC1, MinU1,
                    MinW1),
    minimal_cost(Vs, Matrix, Path, U, W, MinC1, MinU1, MinW1).

row_min(Verteces, Row, W, Cost) :-
    row_min(Verteces, Row, W, Cost, _, 100000).

row_min([], _, W, C, W, C).
row_min([V|Vs], Row, W, C, MinW, MinC) :-
    arg(V, Row, C1),
    update_minimum3(C1, V, MinC, MinW, MinC1, MinW1),
    row_min(Vs, Row, W, C, MinW1, MinC1).


cost(Path, Matrix, Cost) :-
    cost(Path, Matrix, Cost, 0).

cost([_], _, Cost, Cost).
cost([V1,V2|Vs], Matrix, Cost, Cost0) :-
    arg(V1, Matrix, Row),
    arg(V2, Row, C),
    Cost1 is Cost0+C,
    cost([V2|Vs], Matrix, Cost, Cost1).

delete_vertex([], _, []).
delete_vertex([U|Vs], U, Ws) :-
    !,
    delete_vertex(Vs, U, Ws).
delete_vertex([V|Vs], U, [V|Ws]) :-
    delete_vertex(Vs, U, Ws).

update_minimum1(C, _, _, MinC, MinP, MinC1, MinP1) :-
    MinC =< C,
```

```
    !,
    MinC1 = MinC, MinP1 = MinP.
update_minimum1(C, P, V, _, _, C, [V|P]).

update_minimum2(C, _, _, MinC, MinU, MinW, MinC1, MinU1, MinW1) :-
    MinC =< C,
    !,
    MinC1 = MinC, MinU1 = MinU, MinW1 = MinW.
update_minimum2(C, V, W, _, _, _, C, V, W).

update_minimum3(C, _, MinC, MinW, MinC1, MinW1) :-
    MinC =< C,
    !,
    MinC1 = MinC, MinW = MinW1.
update_minimum3(C, V, _, _, C, V).

/************************************************/

% Generating random cost matrix.

random_matrix(N, Matrix) :-
    functor(Matrix, matrix, N),
    empty_rows(N, N, Matrix),
    Limit is (N+N/2)//1,
    randoms_in_matrix(N, N, Limit, Matrix).

empty_rows(0, _, _) :- !.
empty_rows(N, Dim, Matrix) :-
    arg(N, Matrix, Row),
    functor(Row, row, Dim),
    N1 is N-1,
    empty_rows(N1, Dim, Matrix).

randoms_in_matrix(0, _, _, _) :- !.
randoms_in_matrix(I, N, Limit, Matrix) :-
    random_entry(N, I, Limit, Matrix),
    I1 is I-1,
    randoms_in_matrix(I1, N, Limit, Matrix).

random_entry(I, I, _, Matrix) :- !,
    arg(I, Matrix, Row),
    arg(I, Row, 0).
random_entry(J, I, Limit, Matrix) :-
    random(Limit, Random),
    arg(I, Matrix, Row),
    arg(J, Row, Random),
    arg(J, Matrix, Col),
    arg(I, Col, Random),
```

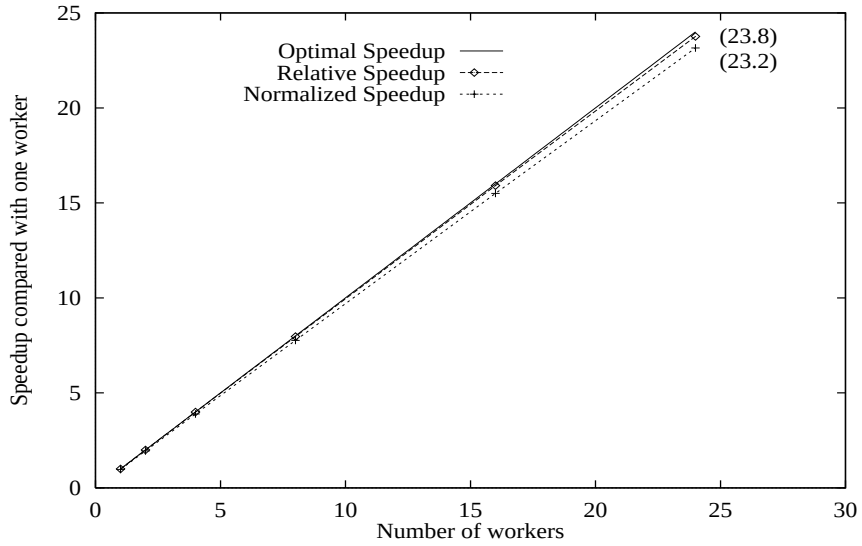
```
    J1 is J-1,
    random_entry(J1, I, Limit, Matrix).

append([],Y,Y).
append([A|X], Y, [A|Z]) :- append(X, Y, Z).
```

Benchmark Plots

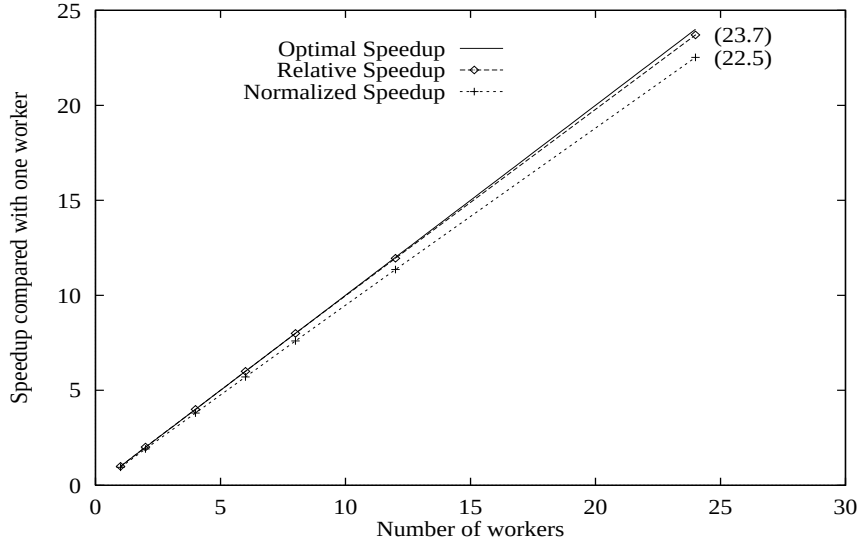
B.1 NREV



B.2 MAP



B.3 COMPARING DNA SEQUENCES



B.4 TRAVELING SALESMAN