**Quintus Proprietary and Confidential**

# Proposal for Parallel Quintus Prolog

Draft 2

1 June 1988

OLD

For ~~Internal~~ Comments

## Table of Contents

## 1. Introduction

Quintus has carefully considered a number of approaches to a parallel version of Quintus Prolog for the Sequent Symmetry™ series of machines, under the Dynix™ version of Unix™. The present proposal indicates intent for a specific direction, among two principal alternatives. After briefly reviewing these alternatives, we indicate why the chosen direction seems to be the most compelling at this time. We then proceed to give details of our approach, from a language viewpoint.

### 1.1. Background

The extension of a sequential language to one which provides for parallel processing is a much-studied topic. There are two main types of approach:

| | |
|---|---|
| Explicit Parallelism | Extend the language with new primitives which set up parallel activities during execution, and which provide for communication and synchronization among these activities. |
| Implicit Parallelism | Do not change the language, but extract, through compilation and dynamic analysis of execution states, opportunities for converting parallel activities to sequential ones. |

Implicit parallelism works best in languages with a strongly declarative component. Prolog is normally thought to be such a language, and indeed, Prolog does facilitate the declarative expression of knowledge. However, Prolog is still, at its roots, procedural rather than declarative. Consequently, it is unreasonable to expect that *all* Prolog code will permit the type of analysis needed for implicit parallelism. Although significant advances have been made in the area of implicitly parallel implementations of Prolog which handle reasonably declarative programs, by admission of researchers working closely in this area, the ideas are not quite ready for commercialization. We elaborate on this view in Appendix I.

Explicit parallelism can be used in virtually any language. What is sometimes not recognized is that explicit parallelism can bring additional *strengths* to a language in certain application domains. For example, implicit parallelism is easiest to manage with "compute-bound" applications, but may be more awkward with "transaction-processing" type applications, where direct interaction with the scheduler can be important.

In the case of Prolog, there is a definite benefit in an explicitly parallel extension based on lightweight processes (called "threads", with the overall scheme being called "multithreading") which we propose below: it enables the implementation of multiple "knowledge servers", which can function as coroutines, each server maintaining its own state of a Prolog query in process[1] The knowledge server concept is similar to the multiple "cursor" facility in SQL. In fact, this benefit is sufficiently great that we intend to incorporate it eventually into the *sequential* version of the product.

The last rationale offered for the election of multithreading is that it does not, in any way,

[1]Here we assume that the reader is familiar with Prolog's capability of returning multiple responses to a query in sequence.

preclude the ultimate introduction of implicit parallelism *within* individual threads; in fact, we are in the process of investigating how this can be staged. The next section proceeds to elaborate on the multithreading concept.

## 2. Broad Characteristics of MultiThreading Primitives

The following are salient aspects of a proposed *multithreading* extension to Quintus Prolog.

- Processes ("threads") are lightweight, in the sense of carrying with them the state of the Prolog engine, rather than an entire Unix process.
- Processes can be created dynamically. There is no *a priori* limit to their number, nor is their any pre-imposed assignment of processes to processors.
- Prolog terms are considered atomic. That is, no attempt is made to have processes bind different variables in a common goal. Rather, a process can be passed one or more parametric terms at creation, but other forms of communication must be done by using either the database, or by explicit channel communication, as discussed below. If the terms contain variables, then a copy of those variables is made; the variables are not shared across processes.
- Processes can create communication channels dynamically. Two or more processes can communicate terms across these channels. If the terms contain variables, then a copy of those variables is made; the variables are not shared across processes.
- Processes share a common Prolog database. This means that they can share compiled code as well as dynamically-asserted rules. Assert, retract, retractall, and abolish are atomic actions on the database. A mutual exclusion mechanism on a predicate prevents overlapping. [At some future time, we will want to add a "test-and-set" style of database modification command, which will permit conditional modification based on current database contents, as well as a transaction facility.]
- Processes can, at the programmer's option, isolate segments of their database. This is through a mutually-agreed discipline, such as making certain modules private to a given process, rather than through an enforcement mechanism.

## 3. Multiprocess Primitives

The following primitives are supported at the language level. We describe the primitives in the manner of the Quintus Prolog language manual: a + before an argument means the argument is input to the command, a - means that the argument is output, and a ? means that it can be either.

| | |
|---|---|
| **fork**(+Goal, -Handle) | Creates a Prolog process with *Goal* as input. *Handle* is a reference to the process, which may be used to set the latter's relative priority. |
| **fork**(+Goal, +Priority, -Handle) | Like the preceding, except that Priority is an integer which specifies the relative priority of the process thus created. |
| **fork**(+Goal) | Like the preceding, except there is no handle nor priority. |
| **make_channel**(-Channel) | Creates a channel object which permits |

| | |
|---|---|
| | communication between two or more processes. |
| **recv**(+Channel, ?Term) | Offers to receive a *Term* from *Channel*, and waits for one to be sent. If several processes are awaiting to receive a term on the same channel, then a FIFO queue of processes is formed, the next term sent being given to the first process on the queue. Other processes await to receive future terms. |
| **send**(+Channel, +Term) | Sends a *Term* on *Channel*. If no process is waiting to receive, then the term is left in a queue in the channel and the process continues. |
| **blocking_send**(+Channel, +Term) | Sends a *Term* on *Channel*, waiting for acceptance by a process doing **recv**. If no process is waiting to receive, then the sending process waits in a queue until the term is received. |

For a given channel, if either the queue of terms to be sent is non-empty or if there is a waiting blocking_send, then we say that the channel has a *send pending*.

| | |
|---|---|
| **await_send**(+List_of_Channels, -Channel) | Waits until one of the channels in List_of_Channels has a send pending. When one does, that channel's identification is returned as Channel. If more than one has a send pending, the first such channel in the list at the time of call will be returned. If more than one channel become pending after a wait is initiated, then the system will choose arbitrarily among those which become pending first. [Note: Coding styles which permit an await_send to "race" against another primitive, such as a **recv** which could nullify the condition satisfying the await_send, should be avoided due to the indeterminacy thus created. |
| **close_channel**(+Channel) | Terminates input to the designated Channel. |
| **get_status**(+Channel, -Status) | Returns the current status of the channel, as an integer. If Status = 0, then the queue of terms is empty and there are no waiting **sends** or **recvs**. If Status > 0, then Status indicates the sum of the number of terms in the queue plus the number of waiting **sends**. If Status < 0, then Status indicates the number of waiting **recvs**. |
| **stop_process** | Terminates the current process. |
| **get_procid**(-Handle) | Gets the handle of the current process. |

| | |
|---|---|
| **set_priority**(+Handle, +Priority) | Set relative priority of process identified by *Handle* to *Priority*. Priorities range over integer values with a maximum and minimum to be determined. Setting a priority to the minimum value is equivalent to suspending the process. |
| **get_priority**(+ProcSpec, -PQspec) why a list?? | Gets process priority. *ProcSpec* is either one Handle or a list of Handles. In the first case, Handle:Priority is returned in *PQspec*. In the second case, a list of Handle:Priority pairs is returned. |

Above, "current process" means the one executing the goal being mentioned.

The usage modes of a channel have some things in common with Ada [4] and Occam [5], however the current Prolog-based model is more dynamic in its means of instantiating processes, and more disciplined in its form of communication. We have simplified channel communication so as not to incur unnecessary inefficiencies in dealing with problems of canceling *rendezvous* [4] requests when a channel has more than one **send** or more than one **recv** request pending. At the same time, it is conjectured that we can perform the most useful types of synchronization and communication using our mechanism.

## 4. Canonical Examples

Here is a simple example of communication using a channel:

| | |
|---|---|
| **producer**(+Channel) | will be a process which produces successive terms on Channel, |
| **filter**(+InChannel, +OutChannel) | will be a process which reads successive terms on InChannel and either passes them to OutChannel or absorbs them, |
| **consumer**(+Channel) | will be a process which reads and uses the successive terms on Channel. |

Note that the modes of these channels are all +, meaning that the channel is supplied to the process from the outside. The mode says nothing about whether terms are sent or received on the channel.

To establish communication in the prescribed way, we might use the following:

```
run :-
    make_channel(Ch1),
    make_channel(Ch2),
    fork(producer(Ch1)),
    fork(filter(Ch1, Ch2)),
    fork(consumer(Ch2)).          what ?
```

For example, suppose pred/1 is a predicate which generates numbers, filter passed only those numbers within a given range, and consumer printed out the numbers. Then we might add pred and range as arguments of producer and filter respectively, the overall code appearing as:

```
run(Pred, Lo, Hi) :-
```

```
        make_channel(Ch1),
        make_channel(Ch2),
        fork(producer(Ch1, Pred)),
        fork(filter(Ch1, Ch2, Lo, Hi)),
        fork(consumer(Ch2)).

producer(Channel, Pred) :-
        call(Pred, X),
        send(Channel, X),
        fail.

filter(In, Out, Lo, Hi) :-
    repeat,
        recv(In, X),
        X >= Lo, X =< Hi,
        send(Out, X),
        fail.

consumer(Channel) :-
    repeat,
        recv(Channel, X),
        write(X), nl,
        fail.
```

what?? Not in language. What to gain by passing in pred??

The reason for the **repeat** sub-goals is to insure looping behavior. This type of construct is probably a bit foreign to the non-Prolog programmer. The **repeat** construct, which is standard to Prolog, provides a "barrier" for backtracking. That is, when a repeat is hit during backtracking, there is always success, so that the goals following repeat will again execute in sequence. The sequence of goals can either be terminated by **fail**, which causes infinite repetition, or by a test which will permit the entire sequence to succeed, thus terminating the iteration achieved by backtracking. The advantage of using it is that *any storage allocated on the run-time stack during the loop body is immediately reclaimed.* this construct (or idiom)

The following expresses the equivalent procedures **filter** and **consumer** using tail-recursion instead fail-loops. failure-driven loops?

```
filter(In, Out, Lo, Hi) :-
        recv(In, X),
        (  (X >= Lo, X =< Hi) -> send(Out, X)
         | true
        ),
        filter(In, Out, Lo, Hi).

consumer(Channel) :-
        recv(Channel, X),
        write(X), nl,
        consumer(Channel).
```

format

The *producer* process cannot be expressed using tail-recursion, since it relies on backtracking to get the solutions of pred/1.

Using similar examples, it is easy to see that various forms of *merging* and *choice service* can be achieved by more than one process sending to, or receiving from, a single channel.

## 5. Detached Goal Solving

In our specification, the **fork** primitive does not permit the transfer of results from the forked process back to the forking process. In fact, some thought reveals that there is no sensible way to do this, since the fork always succeeds, and we have no synchronized access to the variables in the forked goal. However, such a transfer is easily and naturally achieved through the added use of the channel construct. The idea is that we create a channel at the time of forking, for the purpose of receiving the result bindings. Then, by doing **recv** on this channel, we can cause waiting until the goal has been solved.

For greatest overlapped processing on a multiprocessor system, we do this **recv** as late as possible. If, on the other hand, the goal fails, we would like to pass something on the channel to so indicate this information, otherwise the **recv** would wait forever. We adopt the convention that the term **fail** will be sent.

The predicate being described is

**detach**(?Goal, ?ResultForm, -Channel)

Here Goal is the Goal to be solved in parallel, ResultForm is a term which has as its variables a subset of those in Goal, and Channel is a channel that will be created by **detach**. The definition is:

```
detach(Goal, ResultForm, Channel) :-
    make_channel(Channel),
    fork( ((Goal, send(Channel, ResultForm))
         | send(Channel, fail))).
```

The | denotes disjunction, so that the first argument of **fork** reads: if Goal is solved, then send ResultForm (where variable values are filled automatically by virtue of the fact that they are shared between Goal and ResultForm). That is, the definition of forked processes here always makes a *copy* of the goal, and such copying preserves sharing (see the discussion of *structure sharing* in [1]). Thus, the process which solves the conjunction

(Goal, **send**(Channel, ResultForm))

passes the values of any variables which become bound during solving out by appropriate embedding in ResultForm.

The **detach** procedure will usually be followed by a **recv** primitive on the channel thus created. So the typical use is

```
detach(Goal, ResultForm, Channel)
 .
 .
 .
recv(Channel, Result).
```

Using **detach** we can create a more symmetric looking procedure, **par**, which runs its two arguments as goals in parallel.

```
par(Goal1, Goal2) :-
     detach(Goal1, Goal1, C1),
     Goal2,
     recv(C1, Goal1).
```

The result of **par**(Goal1, Goal2) will be to instantiate the variables in Goal1 and Goal2, to the extent that the goals are mutually solvable. Note that if there is a conflict in the binding of the variables, then the **par** goal will fail.

As an example, if we wished to sum in parallel the leaves of a *tree*, where a tree is either a single number, or a list of trees, we could write

```
sum([], 0) :- !.                              % the sum of an empty tree is 0

sum([X | Y], Sum) :-                          % the sum of a non-empty tree
     !,
     par(sum(Y, Ysum), sum(X, Xsum)),         % is done recursively by
     Sum is Xsum + Ysum.                      % summing the sub-trees in parallel

sum(X, X).                                    % the sum of a leaf is the leaf's value
```

Similarly, we mimic this technique in a tree search, looking for leaves which are terms satisfying a given predicate:

```
search([X | Y], Pred, Result) :- !,
     par(search(X, Pred, XResult),
          search(Y, Pred, YResult)),
     or(XResult, YResult, Result).

search(X, Pred, Y) :-
     Term =.. [Pred, X],
     ( (call(Term), Y = X) | Y = fail).

or(A, B, C) :-
    A = fail  -> (C = B)
   | otherwise  -> (C = A).        ) format
```

## 6. Higher-Level Primitives

Even in a single processor environment, having multiple processes provides a means for clean separation of tasks that work together as co-routines. It also provides an easy way to respond to user input when it occurs, while continuing to process ready tasks during "think time". Each process tends to its own task, and its state is preserved when it is suspended.

In a multiple processor, shared memory environment, different processes can be executed simultaneously. If they share the same database of clauses, updates to the database must be serialized. However, because process state persists when the process is suspended, *process structuring reduces the need to update the database temporarily* during a computation to "remember" facts; instead a process can be assigned to remember them. Each process' state is private (the state is contained, for example, in the arguments of a tail-recursive procedure, not the database), so updates to its state need not be serialized with updates to other process' state or the global database.

As a tutorial example, in Prolog, a global counter is frequently used to ensure unique ids (i.e. we have an implementation of the equivalent of *gensym* in Lisp. When an id is needed, an "id server" procedure is called. Its code might be

```
genId(N) :-
    retract(lastId(M)),
    N is M+1,
    asserta(lastId(N)).
```

This code contains two updates of predicate **lastId** in the global database. In contrast, a process (instead of *procedure*) would have code something like

```
genId(M, IdChannel) :-
    blocking_send(IdChannel, M),
    N is M+1,
    genId(N, IdChannel).
```

which does not touch the global database at all. Whenever a process wants an id it does a **recv** on the channel.

The ability of the language to express computations naturally in a way that reduces the bottleneck of global updating is important in achieving substantial performance gains with multiple processors, and can lead to cleaner programs even with a single processor.

## 6.1. A Term Server

An example of a higher-level primitive implementable using the low-level primitives above is a generic "term-server" facility. A special case was the id-server mentioned earlier. We design this primitive to be similar to Prolog's **findall**:

**findall**(ResultForm, Goal, Result)

generates as Result the *list* of terms of the form *ResultForm* which are instantiated results of *Goal*. A process-oriented analog is

**serve**(ResultForm, Goal, Channel)

Instead of generating the solution terms all at once in a list, this creates a process which serves them one-at-a-time via a channel. The programmer does not need know that there is parallelism, and could make use of this primitive even if there were none, since the state of the serving computation can be kept inside of the process. Unlike *findall*, this goal makes sense even if the bag of solutions is *infinite*. It is therefore similar to "lazy-evaluation" in parallel functional languages [6].

As an example, suppose we have the following rules and facts in the database:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), ancestor(Z, Y).

parent(fred, carol).
parent(tom, sam).
parent(fred, mary).
parent(john, fred).
parent(john, tom).
parent(tom, judy).
```

If we created the process

**fork**(**serve**(X, ancestor(john, X), Channel), _)

then on each **recv** from Channel, we would get successive ancestors of john:

fred, tom, carol, mary, sam, judy

To see how the term-server is implemented using the proposed extension, we need only exhibit the coding:

```
serve(Form, Goal, Channel) :-
    call(Goal),
    blocking_send(Channel, Form),
    fail.

serve(Form, Goal, Channel) :-
    send(Channel, end_of_file),
    stop_process.
```

had called sending "fail" a convention - maybe that is too strong a word?

This works by means of a fail-loop. The Goal is called repeatedly, and each time it succeeds, the current instantiation of Form is sent on Channel. When there are no more instantiations, end_of_file is sent by virtue of the second clause.

cap ?

## 6.2. Doall and map

The predicate **doall** attempts to solve a *list* of goals in parallel, where each Goal is accompanied by a ResultForm for packaging the resulting bindings.

```
doall([]).
doall([Goal | More]) :-
    detach(Goal, Goal, Chan),
    doall(More),
    recv(Chan, Goal).
```

The idea here is that a process is created for each goal in the argument list. The recursive clause creates a detached process for the first goal in the list, which can run in parallel with the recursive call of **doall** on the rest of the list. Only after that call is done will waiting for the first goal occur. In this way, the maximal amount of parallel execution is generated.

**doall** thus provides a limited form of *AND-parallelism* [2], but one capable of generating large speedups due to solving each element of a large list of goals in parallel. The technique can similarly be applied to other useful forms, such as **map**, in an obvious way.

```
map(Pred, [], []).
map(Pred, [A|X], [B|Y]) :-
   Term =.. [Pred, A, B],
   detach(Term, Term, Chan),
   map(Pred, X, Y),
```

**recv**(Chan, Term).

For example,

map(parent, [fred, tom, john], X)

gives the result [carol,sam,fred] when run with the *parent* database in the previous section.

Note that we are not providing for backtracking within AND-parallelism; this is up to the user to arrange. Although more conservative in terms of compiler technology, we conjecture that this method of intentional *wholesale* parallelism will often be more fruitful than that provided by smaller scale automatic detection, which is known to still be fraught with problems [3].

Although the primitives developed can also achieve *OR-parallelism* by having multiple processes feed a single channel, in order to get OR-parallel execution of Prolog clauses, we would have to build a meta-interpreter around the user program. This is the means by which one would build a parallel **findall** procedure. We do not show the details here.

## I. Commercial Impact of Implicit Parallel Implementations

### I.1. Types of Implicit Parallelism

Many designs enabling Prolog systems to exploit global shared memory hosts have been proposed by the Logic Programming community. In choosing to implement an industrial-quality "parallel Prolog" within a limited time-frame, to the required standard (i.e. that of Quintus Prolog), it is important to pick a design that meshes well with other related products and tools.

hmm

Implicit parallelism in Prolog implementations is coarsely divisible into "AND-parallel" and "OR-parallel". The first refers to attempting to solve several sub-goals within a given goal simultaneously (corresponding to an "AND" node of a search tree), while the second refers to attempting to solve a given goal in several different ways simultaneously (corresponding to an "OR" node of the tree).[2]

This appendix discusses the status of research on OR-parallel Prolog systems and tries to determine whether a commercial OR-parallel Prolog for a global shared memory machine is a viable short-term option. (We take "short-term" to mean a beta-test within 1 to 1.5 years.) A similar discussion can be developed for AND-parallel Prolog, and we have similarly thought this through, but do not present a comparable argument here.

AND-options are in worse shape...

The discussion is based on close acquaintance with current relevant research, as well as the opinions expressed by key researchers. All designs considered will be assumed equally appropriate for the hosts in question, so hardware and O/S issues can be ignored.

### I.2. The Status of Work in OR-parallel Prolog

The body of research into OR-parallel Prologs owes much to the efforts of the Gigalips Consortium, a cooperation between a group at Manchester University headed by David H. D. Warren, the Swedish Institute of Computer Science (SICS), and researchers at Argonne National Laboratory (ANL). One of the projects of the Gigalips group is Aurora Prolog, a high-quality testbed for OR-parallel Prolog systems based on SICStus Prolog. Because Aurora Prolog, in our opinion, is the most advanced OR-parallel Prolog research environment in existence, we will take it as the state of the art and discuss its implications for a commercial OR-parallel Prolog.

To make clear the differences between Aurora Prolog and a commercial OR-parallel Prolog a technical summary of the former is needed. Aurora Prolog consists, notionally and implementationally, of two main modules. One module, known as a *worker*, constitutes an emulator for the *Warren Abstract Machine* (WAM) that has been modified to operate some of its runtime areas in global memory, and maintain additional structure necessary to avoid binding conflicts for shared variables. A worker traverses the Prolog search tree in OR-parallel under direction of a *scheduler*, the second module. When a Prolog program is to be run in OR-parallel, multiple WAM/scheduler pairs are initialized and set running on stacks in shared memory.

[2]We mention that "searching" is a metaphor often used in describing Prolog systems, but that such systems are basically *computational*; the implied tree is usually never constructed explicitly, except perhaps in debugging via an interpreter.

Scheduling decisions are made by individual workers on the basis of local information in the shared memory. There is no global coordination of workers. Behavior and performance of the workers can be monitored using a suite of tools.

The interface between a WAM and its scheduler is well-defined and documented, which should greatly ease future research (it has already proved valuable to the Gigalips participants). The interface has the advantage that WAM modifications or scheduler changes can be made independently of the other module. It also makes possible debugging tools like a test harness for schedulers conforming to the interface, making it possible to extensively test a scheduler without a working WAM module.

As of the March, 1988 meeting of the Gigalips group in Manchester, a single version of the modified SICStus WAM was being used in all investigations. Variable bindings are handled using the *SRI model* first proposed by David H. D. Warren in 1983. The three main Gigalips participants are, at least for the time being, committed to the SRI model, although some work is being done by associated groups on alternatives. The Gigalips group's satisfaction with the SRI model represents a significant development for commercial prospects, as will be discussed later.

Two schedulers, one done at ANL and one done at Manchester, were being tested in Manchester, and a third, very promising, scheduler design was proposed by a researcher from SICS. While the existing schedulers handled pure (e.g. cut-free and side-effect-free) Prolog acceptably, they remain a long way from running arbitrary Prolog programs. Most of the research spent on Aurora Prolog this March went into scheduler issues.

might want to tone this down in light of the SICS paper.

## I.3. Moving from Aurora Prolog to a Commercial OR-parallel Prolog

Commercialization of each of the two modules of Aurora Prolog will be discussed separately. The major issues involved in the design and implementation of the WAM module of a commercial OR-parallel Prolog can be split into the areas of emulator support and memory management. Gigalips research has demonstrated that degradation of sequential Prolog performance due to the SRI model can be made acceptably low. Sequential performance of an OR-parallel Prolog system is important as it defines the amount of parallelism that MUST be exploited ~~achieved~~ to come out ahead in the race with fast sequential Prologs. Preliminary studies at Quintus also indicate that a modified Quintus Prolog emulator should yield reasonable sequential performance.

The Aurora system does not adequately address memory management issues. It has not been designed to be very careful with memory (e.g. implementing stack expansion using UNIX's realloc(3)). However, independent of the "parallel Prolog" paradigm chosen, the memory management rewrite undertaken for the next major release of Quintus Prolog would go a long way toward efficiently managing multiple scheduler-WAM invocations in a single address space.

The real remaining issue for the WAM module is the dependence of an OR-parallel Prolog using the SRI model on scheduling technology. The main drawback of the SRI model, which has not yet been adequately assessed, is that the cost of a worker switching tasks is proportional to the distance of the new task from the old. ("Distance" can roughly be measured in terms of the amount of WAM trail not shared between the old and new positions of the worker on the search tree.) Thus, when using the SRI model, it may be essential to good performance that the scheduler be smart enough to find the best available work. If this cannot be done, or if programs require long-distance relocation of workers, variable binding models may have to be reconsidered.

Scheduler issues point to the maturation still needed by Aurora Prolog. One of the motivations for investigations of OR-parallel logic programming languages rather than, say, committed choice languages like GHC or FCP, is the prospect of retaining the Prolog language. The only difference (we might hope!) would be that applications would run faster. Unfortunately, efficiently executing arbitrary Prolog programs in OR-parallel means having to deal with problems that are often not encountered when demonstrating a design concept. The problems, in general, have to do with suspending and restarting workers either as desired by the semantics of the language (in the case of cut/commit and predicates with side-effects, or when retaining the Prolog order of solutions) or for better efficiency (e.g. if a branch suspends, its worker might detach itself and search for work elsewhere).

Scheduling issues also find their way into the language. At least three forms of cut (the standard Prolog cut and two forms of commit) are being considered. The three have different scheduling demands, with the normal Prolog cut giving the most difficulty. Various proposals have been aired as to how cuts can be implemented, however, all complicate scheduling significantly. One Gigalips group (ANL) has proposed not even attempting to handle arbitrary Prolog programs yet, but instead to try to efficiently execute programs in which parallel sections are circumscribed and contain only a subset of Prolog. Under this view, a parallel section of a program must not cause side-effects (e.g. do I/O or assert/retract), cannot care about solution order, and make do with a restricted notion of cut. The point is that such restrictions on the language make it possible to implement a usable system with scheduler technology that is understood today.

Other proposals include a first attempt at using annotations to classify procedures into, for instance, those not needing Prolog's solution ordering. Some techniques as this might eventually help to ease the scheduling burden.

It is Quintus' view that better performance at the cost of significant departures from the Prolog language is not in the application developer's best interest. Our experience shows that compatibility across platforms is very important - our product line reflects our appreciation of this fact. It is probably not in the host machine vendor's interest either, as it will make applications difficult to port to or from other machines. The implementation would be more likely used for experimentation than application delivery, making it less attractive in terms of sales. In addition, because research into more general solutions is active, such a system would quickly become obsolete. Thus, we would not find a strategy like ANL's acceptable in a commercial product.

The alternative is to implement a system now that runs arbitrary Prolog programs in OR-parallel. As we have seen, however, research into scheduling in the context of side-effects, cuts and frequent worker interruptions, even only to be correct, is really just beginning. Estimates of finally achievable performance should also be looked at with the scheduler in mind. If it is important that schedulers choose work intelligently to avoid task switching overheads, especially in an execution environment in which interrupts are frequent due to side-effects and solution ordering, the scheduler becomes grossly more complicated.

tone down

It appears that man-years are being committed [illegible] by the Gigalips groups to scheduler problems. (While Quintus could also undertake scheduler research, there is no reason to believe that we would be able to achieve results faster than the Gigalips participants.) When we take into consideration that good scheduling may be essential to efficient execution of programs on OR-parallel, we must conclude that, at least in the short term, a commercial OR-parallel Prolog is not viable.

## II. Resource Estimates and Time-Table

The following is a resource estimate for the development of the multithreading version of Prolog on the Sequent. Except as noted, the numbers are not yet firm, although every effort will be made to meet this schedule, at a minimum.

### II.1. New Memory Management

In order to make multithreading possible, a re-design of the current memory management system is necessary. Following are the principal tasks involved, the resource estimates, and target dates.

1. Block server and initialization of code-space and stacks from environment variables working (but not integrated into system). [3 man-months, done]
2. Elevate the above into a working Prolog WAM system. No stack shifter or garbage collector. Runs test programs. [1 man-month, done]
3. Add stack shifter. [0.5 man-month, 15 June, 1988]
4. Garbage collector working. [2 man-months, 15 September, 1988 (gap due to Vacation and staffing changes)]
5. Interim testing and debugging. [0.5 man-month, 30 September, 1988]
6. Porting memory-management concepts to Sequent. [2 man-months, 28 November, 1988]

### II.2. Multithreading State Handling

Approximately 12 man-months are estimated to fully design implement the needed state-switching mechanisms and multithreading primitives. A detailed plan has not been developed as yet. With anticipated loading, this phase will be complete mid-May, 1989.

### II.3. Scheduler and Sequent Operating System Calls

Approximately 7 man-months are estimated to provide operating-system support to the multithreading primitives and scheduler under the Sequent operating system. Documentation will be done during this period. A detailed plan has not been developed as yet. With anticipated loading, this phase will be complete by the end of July, 1989.

### II.4. Beta Test

Assuming a two-month beta-test period, this phase will be completed by the end of September, 1989.

## References

[1] R.S. Boyer and J.S. Moore.
The sharing of structure in theorem proving programs.
*Machine Intelligence* 7:101-116, 1972.
Edinburgh University Press, Scotland.

[2] J.S. Conery.
*The AND/OR process model for parallel interpretation of logic programs.*
Technical Report 204, U.C. Irvine, June, 1983.

[3] D. DeGroot.
Restricted and-parallelism.
In *International Conference on Fifth generation computer systems*, pages 471-478.
ICOT, 1984.

[4] U.S. Department of Defense.
*Reference manual for the Ada programming language*
ANSI/MIL-STD 1815 A edition, 1983.

[5] Inmos.
*Transputer reference manual*
Inmos Limited, Bristol, England, 1985.

[6] R.M. Keller and F.C.H. Lin.
Simulated performance of a reduction-based multiprocessor.
*Computer* 17(7):70-82, July, 1984.