



UNIVERSITY OF AMSTERDAM

UvA-DARE (Digital Academic Repository)

Logic programming for knowledge-intensive interactive applications

Wielemaker, J.

Publication date
2009

[Link to publication](#)

Citation for published version (APA):

Wielemaker, J. (2009). *Logic programming for knowledge-intensive interactive applications*. [Thesis, fully internal, Universiteit van Amsterdam].

General rights

It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations

If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: <https://uba.uva.nl/en/contact>, or a letter to: Library of the University of Amsterdam, Secretariat, P.O. Box 19185, 1000 GD Amsterdam, The Netherlands. You will be contacted as soon as possible.

Chapter 6

Native Preemptive Threads in SWI-Prolog

About this chapter This chapter was published at the ICLP-03 (Wielemaker 2003a). It describes adding multi-threading to Prolog, a requirement for creating scalable web services and therefore part of research question 2. Threads also play a vital role in the mediator-based MVC design that we used to implement Triple20 (chapter 2, Wielemaker et al. 2005). The SWI-Prolog implementation is the basis for the ISO standard for adding threads to Prolog (Moura et al. 2008) and has already been implemented by two major Open Source Prolog systems (YAP and XSB). Section 6.6.1 has been updated to reflect enhancements made since this paper was published.

Abstract Concurrency is an attractive property of a language to exploit multi-CPU hardware or perform multiple tasks concurrently. In recent years we see Prolog systems experimenting with multiple threads only sharing the database. Such systems are relatively easy to build and remain close to standard Prolog while providing valuable extra functionality. This article describes the introduction of multiple threads in SWI-Prolog exploiting OS-native support for threads. We discuss the extra primitives available to the Prolog programmer as well as implementation issues. We explored speedup on multi-processor hardware and speed degradation when executing a single task.

6.1 Introduction

There are two approaches to concurrency in the Prolog community, implicit fine-grained parallelism where tasks share Prolog variables and implementations (see section 6.7) in which Prolog engines only share the database (clauses) and run otherwise completely independent. We call the first class *parallel* logic programming systems and the latter *multi-threaded* systems. Writing programs for *multi-threaded* Prolog is close to normal Prolog programming, which makes multi-threading attractive for applications that benefit from coarse grained concurrency. Below are some typical use-cases.

- *Network servers/agents*
Network servers must be able to pay attention to multiple clients. Threading allows multiple, generally almost independent, tasks to make progress at the same time. This can improve overall performance by exploiting multiple CPUs (SMP) or by better utilising a single CPU if (some) tasks are I/O bound. Section 6.4.1 provides an example.
- *Embedding in multi-threaded servers*
Concurrent network-service infrastructures such as CORBA or .NET that embed a single threaded Prolog engine must serialise access to Prolog. If Prolog is responsible for a significant amount of the computation Prolog becomes a bottleneck. Using a multi-threaded Prolog engine the overall concurrent behaviour of the application can be preserved.
- *Background processing in interactive systems*
Responsiveness and usefulness of interactive applications can be improved if background processing deals with tasks such as maintaining *mediators* (section 2.4), spell-checking and syntax-highlighting. Implementation as a foreground process either harms response-time or is complicated by interaction with the GUI event-handling.
- *CPU-intensive tasks*
On SMP systems CPU-intensive tasks that can easily be split into independent subtasks can profit from a multi-threaded implementation. Section 6.6.2 describes an experiment.

This article is organised as follows: in section 6.2 we establish requirements for multi-threaded Prolog that satisfy the above use cases. In subsequent sections we motivate choices in the design and API. Section 6.5 provides an overview of the implementation effort needed to introduce threads in a single threaded Prolog implementation, where we pay attention to atom garbage collection. We perform two performance analysis: the first explores the performance loss when running single-threaded applications on a multi-threaded system while the second explores speedup when running a CPU-intensive job on multi-CPU hardware. Finally we present related work and draw our conclusions.

6.2 Requirements

Combining the use-cases from the introduction with the need to preserve features of interactive program development in Prolog such as aborting execution and incremental recompilation during debugging, we formulate the following requirements:

- *Smooth cooperation with (threaded) foreign code*
Prolog applications operating in the real world often require substantial amounts of ‘foreign’ code for interaction with the outside world: window-system interface, interfaces to dedicated devices and networks. Prolog threads must be able to call arbitrary

foreign code without blocking the other (Prolog-)threads and foreign code must be able to create, use and destroy Prolog engines.

- *Simple for the Prolog programmer*
We want to introduce few and easy to use primitives to the Prolog programmer.
- *Robust during development*
We want to be as robust as feasible during interactive use and the test-edit-reload development cycle. In particular this implies the use of synchronisation elements that will not easily create deadlocks when used incorrectly.
- *Portable implementation*
We want to be able to run our multi-threaded Prolog with minimal changes on all major hardware and operating systems.

Notably the first and last requirement suggests to base Prolog threads on the POSIX thread API (Butenhof 1997). This API offers preemptive scheduling that cooperates well with all (blocking) operating system calls. It is well supported on all modern POSIX-based systems. On MS-Windows we use a mixture of pthread-win32¹ for portability and the native Win32 thread-API where performance is critical.

6.3 What is a Prolog thread?

A Prolog thread is an OS-native thread running a Prolog *engine*, consisting of a set of stacks and the required state to accommodate the engine. After being started from a *goal* it proves this goal just like a normal Prolog implementation by running predicates from a *shared* program space. Figure 6.1 illustrates the architecture. As each engine has its own stacks, Prolog terms can only be transferred between threads by copying. Both dynamic predicates and FIFO queues of Prolog terms can be used to transfer Prolog terms between threads.

6.3.1 Predicates

By default, all predicates, both static and dynamic, are shared between all threads. Changes to static predicates only influence the test-edit-reload cycle, which is discussed in section 6.5. For dynamic predicates we kept the ‘logical update semantics’ as defined by the ISO standard (Deransart et al. 1996). This implies that a goal uses the predicate with the clause-set as found when the goal was started, regardless of whether clauses are asserted or retracted by the calling thread or another thread. The implementation ensures consistency of the predicate as seen from Prolog’s perspective. Consistency as required by the application such as clause order and consistency with other dynamic predicates must be ensured using *synchronisation* as discussed in section 6.3.2.

¹<http://sources.redhat.com/pthreads-win32/>

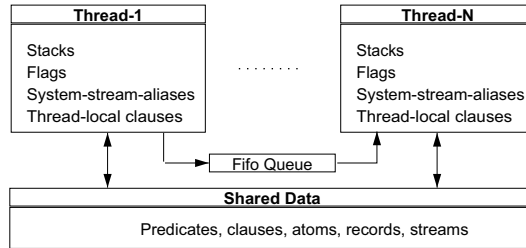


Figure 6.1: Multiple Prolog engines sharing the same database. Flags and the system-defined stream aliases such as `user_input` are copied from the creating thread. Clauses are normally shared, except for thread-local clauses discussed below in section 6.3.1.

In contrast, *Thread-local* predicates are dynamic predicates that have a different set of clauses in each thread. Modifications to such predicates using `assert/1` or `retract/1` are only visible from the thread that performs the modification. In addition, such predicates start with an empty clause set and clauses remaining when the thread dies are automatically removed. Like the related POSIX thread-specific data primitive, thread-local predicates simplifies making code designed for single-threaded use *thread-safe*.

6.3.2 Synchronisation

The most difficult aspect of multi-threaded programming is the need to *synchronise* the concurrently executing threads: ensure they use proper protocols to exchange data and maintain invariants of *shared-data* in dynamic predicates. Given the existence of the POSIX thread standard and our decision to base our thread implementation on this standard for portability reasons, we must consider modelling the Prolog API after it. POSIX threads offer two mechanisms to organise thread synchronisation:

- *A mutex*
is a **Mutual Exclusive** device. At most one thread can ‘hold’ a mutex. By associating a mutex to data it can be assured only one thread has access to this data at one time, allowing it to maintain the invariants.
- *A condition variable*
is an object that can be used to wait for a certain *condition*. For example, if data is not in a state where a thread can start using it, a thread can wait on a condition variable associated with this data. If another thread updates the data it *signals* the condition variable, telling the waiting thread something has changed and it may re-examine the condition.

As Butenhof (1997) explains in chapter 4, the commonly used thread cooperating techniques can be realised using the above two primitives. However, these primitives are not suitable for the Prolog user because great care is required to use them in the proper order and to complete all steps of the protocol. Failure to do so may lead to data corruption or to a deadlock where all threads are waiting for an event to happen that never will. Non-determinism, exceptions and the interactive development-cycle supported by Prolog complicate this further.

Examining other systems (section 6.7), we find a more promising synchronisation primitive in the form of a FIFO (first-in-first-out) queue of Prolog terms. Queues (also called *channels* or *ports*) are well understood, easy to understand by non-experts in multi-threading, can safely handle abnormal execution paths (backtracking and exceptions) and can naturally represent serialised flow of data (*pipeline*). Next to the FIFO queues we support goals guarded by a mutex by means of `with_mutex(Mutex, Goal)` as defined in section 6.4.2.

6.3.3 I/O and debugging

Support for multi-threaded I/O is rather primitive. I/O streams are global objects that may be created, accessed and closed from any thread knowing their handle. All I/O predicates lock a mutex associated with the stream, providing elementary consistency, but the programmer is responsible for proper closing the stream and ensuring streams are not accessed by any thread after closing them.

Stream alias names for the system streams (e.g., `user_input`) are *thread-specific*, where a new thread starts with the current bindings in its creator. Local system stream aliases allow us to re-bind the user streams and provide separate interaction consoles for each thread as implemented by `attach_console/0`. The console is realised using a clone of the normal SWI-Prolog console on Windows or an instance of the `xterm` application in Unix. The predicate `interactor/0` creates a thread, attaches a console and runs the Prolog toplevel.

Using `thread_signal/2` to execute `attach_console/0` and `trace/0` in another thread, the user can attach a console to any thread and start the debugger in any thread as illustrated in figure 6.2.

6.4 Managing threads from Prolog

An important requirement is to make threads easy for the programmer, especially for the task we are primarily targeting at, interacting with the outside world. First we start with an example, followed by a partial description of the Prolog API and the consequences for the foreign language interface.

6.4.1 A short example

Before describing the details, we present the implementation of a simple network service in figure 6.3. We will not discuss the details of all built-in and library predicates used in this

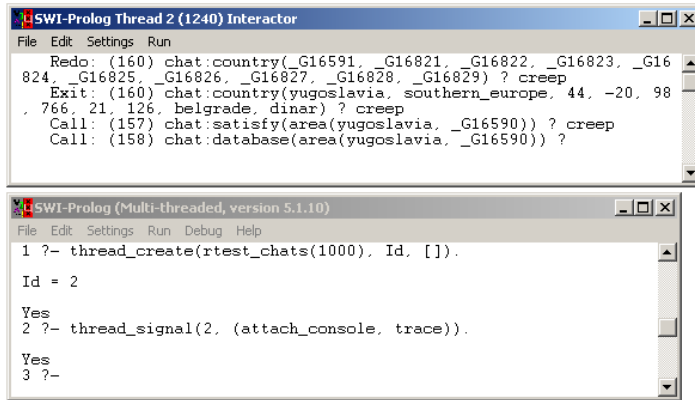


Figure 6.2: Attach a console and start the debugger in another thread.

example. The thread-related predicates are discussed in more detail in section 6.4.2 while all details can be found in the SWI-Prolog reference manual (Wielemaker 2008). Our service handles a single TCP/IP request per connection, using a specified number of ‘worker threads’ and a single ‘accept-thread’. The accept-thread executes `acceptor/2`, accepting connection requests and adding them to the queue for the workers. The workers execute `worker/1`, getting the accepted socket from the queue, read the request and execute `process/2` to compute a reply and write this to the output stream. After this, the worker returns to the queue for the next request.

The advantages of this implementation over a traditional single-threaded Prolog implementation are evident. Our server exploits SMP hardware and will show much more predictable response times, especially if there is a large distribution in the time required by `process/1`. In addition, we can easily improve on it with more monitoring components. For example, `acceptor/2` could immediately respond with an estimated reply time, and commands can be provided to examine and control activity of the workers. Using multi-threaded code, such improvements do not affect the implementation of `process/2`, keeping this simple and reusable.

6.4.2 Prolog primitives

This section discusses the main features of built-in predicates we have added to Prolog to facilitate threads. A full description is in the SWI-Prolog reference manual (Wielemaker 2008).

thread_create(:Goal, -Id, +Options)

Create a thread which starts executing *Goal*. *Id* is unified with the thread-identifier. In

<pre> :- use_module(library(socket)). make_server(Port, Workers) :- create_socket(Port, S), message_queue_create(Q), forall(between(1, Workers, _), thread_create(worker(Q), _, [])), thread_create(acceptor(S, Q), _, []). create_socket(Port, Socket) :- tcp_socket(Socket), tcp_bind(Socket, Port), tcp_listen(Socket, 5). </pre>	<pre> acceptor(Socket, Q) :- tcp_accept(Socket, Client, _Peer), thread_send_message(Q, Client), acceptor(Socket, Q). worker(Q) :- thread_get_message(Q, Client), tcp_open_socket(Client, In, Out), read(In, Command), close(In), process(Command, Out), close(Out), worker(Q). process(hello, Out) :- format(Out, 'Hello world!\n', []). </pre>
---	---

Figure 6.3: Implementation of a multi-threaded server. Threading primitives are set in bold. The left column builds the server. The top-right runs the *acceptor* thread, while the bottom-right contains the code for a *worker* of the crew.

the calling thread, `thread_create/3` returns immediately. The new Prolog engine runs independently. Threads can be created in two modes: *attached* and *detached*. Completion of *Attached* threads must be followed by a call to `thread_join/2` to retrieve the result-status and reclaim all resources. *Detached* threads vanish automatically after completion of *Goal*. If *Goal* terminated with failure or an exception, a message is printed to the console. *Options* is an ISO option list providing the mode, a possible alias name and runtime parameters such as desired stack limits.

thread_join(+Id, -Result)

Wait for the thread *Id* to finish and unify *Result* with the completion status, which is one of `true`, `false` or `exception(Term)`.

message_queue_create(-Queue, +Options)

Create a FIFO message queue (*channel*). Message queues can be read from multiple threads. Each thread has a message queue (*port*) attached as it is created. *Options* allows naming the queue and define a maximum size. If the queue is full, writers are suspended.

thread_send_message(+QueueOrThread, +Term)

Add a copy of *term* to the given queue or default queue of the thread. Suspends the caller if the queue is full.

thread_get_message([+Queue], ?Term)

Get a message from the given queue (*channel*) or default queue if *Queue* is omitted (*port*). The first message that unifies with *Term* is removed from the queue and returned. If multiple threads are waiting, only one will be given the term. If the queue has no matching terms, execution of the calling thread is suspended.

with_mutex(+Name, :Goal)

Execute *Goal* as *once/1* while holding the named mutex. *Name* is an atom. Explicit use of mutex objects is used to serialise access to code that is not designed for multi-threaded operation as well as coordinate access to shared dynamic predicates. The example below updates *address/2*. Without a mutex another thread may see no address for *Id* if it executes just between the *retractall/1* and *assert/1*.

```
set_address(Id, Address) :-
    with_mutex(address, (retractall(address(Id, _)),
                        assert(address(Id, Address)))).
```

thread_signal(+Thread, :Goal)

Make *Thread* execute *Goal* on the first opportunity, i.e., run *Goal* in *Thread* as an *interrupt*. If *Goal* raises an exception, this exception is propagated to the receiving thread. This ability to raise an exception in another thread can be used to abort threads. See below. Long running and blocking foreign code may call `PL_handle_signals()` to execute pending signals and return control back to Prolog if `PL_handle_signals()` indicates that the handler raised an exception by returning -1.

Signalling threads is used for debugging purposes (figure 6.2) and ‘manager’ threads to control their ‘work-crew’. Figure 6.4 shows the code of both the worker and manager needed to make a worker stop processing the current job and obtain a new job from a queue. Figure 6.6 shows the work-crew design pattern to which this use case applies.

Worker	Manager
<code>worker(Queue) :-</code> <code>thread_get_message</code> (Queue, Work), <code>catch</code> (do_work(Work), <code>stop</code> , cleanup), worker(Queue).	... <code>thread_signal</code> (Worker, <code>throw(stop)</code>), ...

Figure 6.4: Stopping a worker using `thread_signal/2`. Bold fragments show the relevant parts of the code.

6.4.3 Accessing Prolog threads from C

Integration with foreign (C-)code has always been one of the main design goals of SWI-Prolog. With Prolog threads, flexible embedding in multi-threaded environments becomes feasible. We identify three use cases. Some applications in which we want to embed Prolog use few threads that live long, while others frequently created and destroy threads and finally, there are applications with large numbers of threads.

Compared to POSIX threads in C, Prolog threads use relatively much memory resources and creating and destroying a Prolog thread is relatively expensive. The first class of applications can associate a Prolog thread (engine) with every thread that requires Prolog access (1-to-1-design, see below), but for the other two scenarios this is not desirable and we developed an N -to- M -design:

- *1-to-1-design*

The API `PL_thread_attach_engine()` creates a Prolog engine and makes it available to the thread for running queries. The engine may be destroyed explicitly using `PL_thread_destroy_engine()` or it will be destroyed automatically when the underlying POSIX thread terminates.

- *N-to-M-design*

The API `PL_create_engine()` creates a Prolog engine that is not associated to any thread. Multiple calls can be used to create a pool of M engines. Threads that require access to Prolog claim and release an engine using `PL_set_engine()`. Claiming and releasing an engine is a fast operation and the system can realise a suitable pool of engines to balance concurrency and memory requirements. A demo implementation is available.²

6.5 Implementation issues

We tried to minimise the changes required to turn the single-engine and single-threaded SWI-Prolog system into a multi-threaded version. For the first implementation we split all global data into three sets: data that is initialised when Prolog is initialised and never changes afterwards, data that is used for shared data-structures, such as atoms, predicates, modules, etc. and finally data that is only used by a single engine such as the stacks and virtual machine registers. Each set is stored in a single C-structure, using thread-specific data (section 6.3.2) to access the engine data in the multi-threaded version. Update to shared data was serialised using mutexes.

A prototype using this straight-forward transition was realised in only two weeks, but it ran slowly due to too heavy use of `pthread_getspecific()` and too many mutex synchronisation points. In the second phase, fetching the current engine using `pthread_getspecific()` was reduced by caching this information inside functions that use it multiple times and passing

²<http://gollem.science.uva.nl/twiki/pl/bin/view/Development/MultiThreadEmbed>

it as an extra variable to commonly used small functions as identified using the `gprof` (Graham et al. 1982) profiling tool. Mutex contention was analysed and reduced from some critical places:³

- All predicates used reference counting to clean up deleted clauses after `retract/1` for dynamic or (re-)consult/1 for static code. Dynamic clauses require synchronisation to make changes visible and cleanup erased clauses, but static code can do without this. Reclaiming dead clauses from static code as a result of the test-edit-recompile cycle is left to a garbage collector that operates similarly to the atom garbage collection described in section 6.5.1.
- Permanent heap allocation uses a pool of free memory chunks associated with the thread's engine. This allows threads to allocate and free permanent memory without synchronisation.

6.5.1 Garbage collection

Stack garbage collection is not affected by threading and continues concurrently. This allows for threads under real-time constraints by writing them such that they do not perform garbage collections, while other threads can use garbage collection.

Atom garbage collection is more complicated because atoms are shared global resources. Atoms referenced from global data such as clauses and records use reference counting, while atoms reachable from the stacks are marked during the marking phase of the atom garbage collector. With multiple threads this implies that all threads have to mark their atoms before the collector can reclaim unused atoms. The pseudo code below illustrates the signal-based implementation used on Unix systems.

```
atom_gc()
{ mark_atoms_on_stacks();           // mark my own atoms
  foreach(thread except self)       // ask the other threads
  { pthread_kill(thread, SIG_ATOM_GC);
    signalled++;
  }
  while(signalled-- > 0)           // wait until all is done
    sem_wait(atom_semaphore);
  collect_unmarked_atoms();
}
```

A thread receiving `SIG_ATOM_GC` calls `mark_atoms_on_stacks()` and signals the `atom_semaphore` semaphore when done. The `mark_atoms_on_stacks()` function is designed such that it is safe to call it asynchronously. Uninitialised variables on the Prolog stacks may be interpreted incorrectly as an atom, but such mistakes are infrequent and can be corrected in a later run of the garbage collector. The atom garbage collector holds the

³See *Update* at the end of section 6.6 for additional places where we reduced contention.

atom mutex, preventing threads to create atoms or change the reference count. The marking phase is executed in parallel.

Windows does not provides asynchronous signals and synchronous (cooperative) marking of referenced atoms is not acceptable because the invoking thread as well as any thread that wishes to create an atom must block until atom GC has completed. Therefore the thread that runs the atom garbage collector uses `SuspendThread()` and `ResumeThread()` to stop and restart each thread in turn while it marks the atoms of the suspended thread.

Atom-GC and GC interaction SWI-Prolog uses a sliding garbage collector (Appleby et al. 1988). During the execution of GC, it is hard to mark atoms. Therefore during atom-GC, GC cannot start. Because atom-GC is such a harmful activity, we should avoid it being blocked by a normal GC. Therefore the system keeps track of the number of threads executing GC. If GC is running in some thread, atom-GC is delayed until no thread executes GC.

6.5.2 Message queues

Message queues are implemented using POSIX condition variables using the recorded database for storing Prolog terms in the queue. Initially the implementation used the `pthreads-win32` emulation on Windows. In the current implementation this emulation is replaced by a native Windows alternative (Schmidt and Pyarali 2008) which does not comply fully to the POSIX semantics for condition variables, but provides an approximately 250 times better throughput of messages. The incorrectness has no implications for our purposes.

The SWI-Prolog recorded database cooperates with the atom garbage collector using atom reference counts: recording a term increments the reference count of each atom that appears in it and erasing the record decrements the reference counts. In both message queues and `findall/3`, recorded terms fulfil the role of temporary storage and the need to synchronise twice for every atom in a term has proven to be a major performance bottleneck. In the current implementation, atoms in temporary recorded terms are no longer registered and unregistered. Instead, atom garbage collection marks atoms that appear in queued records and the solution store of `findall/3`. See also the *update* paragraph at the end of section 6.6.

6.6 Performance evaluation

Our aim was to use the multi-threaded version as default release version, something which is only acceptable if its performance running a normal non-threaded program is close to the performance of the single-threaded version, which is investigated in section 6.6.1. In section 6.6.2 we studied the speedup on SMP systems by splitting a large task into subtasks that are distributed over a pool of threads.

6.6.1 Comparing multi-threaded to single threaded version

We used the benchmark suite by Fernando Pereira⁴ for comparing the single threaded to the multi threaded version. The Pereira benchmark set consists of 34 tests, each of which tests a specific aspect of a Prolog implementation. The tests themselves are small and iterated many times to run for a measurable time. We normalised the iterations of each test to make it run for approximately one second on our base case: single-threaded SWI-Prolog on Linux. We ran the benchmarks in five scenarios on the same hardware, a machine with an 550Mhz Crusoe CPU running SuSE Linux 7.3 and Windows 2000 in dual-boot mode.

Bar	Threading	OS	Comments
1	Single	Linux	Our <i>base-case</i> .
2	Single	Linux	With extra variable. See description.
3	Multi	Linux	Normal release version.
4	Single	Windows	Compiled for these tests.
5	Multi	Windows	Normal release version.

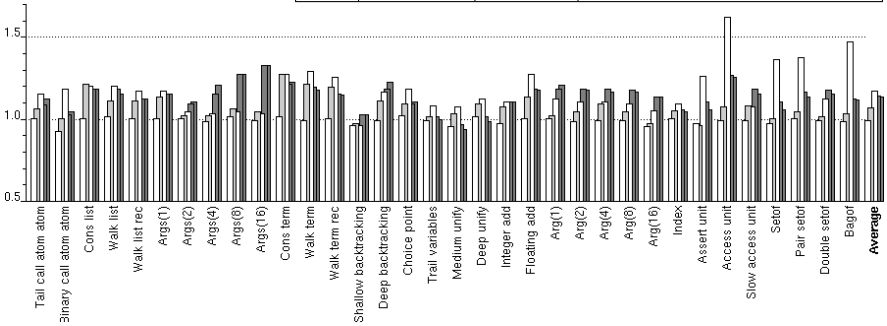


Figure 6.5: Performance comparison between single and multi-threaded versions. The Y-axis shows the time to complete the benchmark in seconds. The legend above summarises the test setting represented by each bar. The rightmost 5 bars show the average.

Figure 6.5 shows the results. First, we note there is no significant difference on any of the tests between the single- and multi-threaded version on Windows 2000 (bars 4&5). The figure does show a significant difference for Linux running on the same hardware (bars 1&3). Poor performance of Linux on ‘assert unit’, ‘access unit’ and the setof/bagof tests indicates a poor implementation of mutexes that are used for synchronising access to dynamic predicates and protecting atoms in records used for storing the setof/bagof results.

The slow-down on the other tests cannot be explained by synchronisation primitives as they need no synchronisation. The state of the virtual machine in the single threaded version is stored in a global structure, while it is accessible through a pointer passed between functions in the multi threaded version. To explain the differences on Linux we first compiled a

⁴<http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/lang/prolog/code/bench/pereira.txt>

version that passes a pointer to the virtual machine state but is otherwise identical to the single threaded version. This version (2nd bar) exhibits behaviour similar to the multi-threaded (3th bar) version on many of the tests, except for the tests that require heavy synchronisation. We conclude that the extra variable and argument in many functions is responsible for the difference. This difference does not show up in the Windows version. We see two possible explanations for that. First, Unix shared object management relies on *position independent* code, for which it uses a base register (EBX on IA32 CPUs), while Windows uses *relocation* of DLLs. This implies that Unix systems have one register less for application code, a significant price on a CPU with few registers. Second, because overall performance of the single threaded version is better on Linux, it is possible that overall optimisation of gcc 2.95 is better than MSVC 5 or to put in differently, Windows could have been faster in single threaded mode if the optimisation of MSVC 5 would have been better.

Finally, we give the average result (last column of figure 6.5) comparing the single-threaded with the multi-threaded version (cases 1 and 3 in figure 6.5) for a few other platforms and compilers. Dual AMD-Athlon, SuSE 8.1, gcc 3.1: -19%; Single UltraSPARC, Solaris 5.7, gcc 2.95: -7%; Single Intel PIII, SuSE 8.2, gcc 3.2: -19%. Solaris performs better on the mutex-intensive tests.

Update We re-ran the Linux comparison on modern hardware: AMD Athlon X2 5400+ dual core CPU running Linux 2.6.22, glibc 2.6.1 and gcc 4.2.1. Both single and multi-threaded versions were compiled for the AMD64 (x86_64) architecture, which provides 16 instead of 8 general purpose registers. Since publication of this paper we changed the implementation of `findall/3` to use the variant of recorded terms described in section 6.5.2 to avoid the need for synchronisation in the all-solution predicates.

The average performance loss over the 34 tests is now only 4%. This is for a large part caused by the dynamic predicate tests (`assert_unit` and `access_unit`; -20% and -29%). Because `findall/3` and friends no longer require significant synchronisation, the multi-threaded version performs practically equal to the single-threaded version on the last four tests of figure 6.5 (`setof ... bagof`).

6.6.2 A case study: Speedup on SMP systems

This section describes the results of multi-threading the Inductive Logic Programming system Aleph (Srinivasan 2003), developed by Ashwin Srinivasan at the Oxford University Computing Laboratory. Inductive Logic Programming (ILP) is a branch of machine learning that synthesises logic programs using other logic programs as input.

The main algorithm in Aleph relies on searching a space of possible general clauses for the one that scores best with respect to the input logic programs. Given any one example from the input, a lattice of plausible single-clauses ordered by generality is bound from above by the clause with `true` as the body (\top), and bound from below by a long (up to hundreds of literals) clause known as the most-specific-clause (or bottom, \perp) (Muggleton 1995).

Many strategies are possible for searching this often huge lattice. Randomised local search (Železný et al. 2003) is one form implemented in Aleph. Here a node in the lattice is selected at random as a starting location to (*re*)-start the search. A finite number of *moves* (e.g., radially from the starting node) are made from the start node. The best scoring node is recorded, and another node is selected at random to restart the search. The best scoring node from all restarts is returned.

As each restart in a randomised local search of the lattice is independent, the search can be multi-threaded in a straight forward manner using the worker-crew model, with each worker handling moves from a random start point and returning the best clauses as depicted in figure 6.6. We exploited the thread-local predicates described section 6.3.1 to make the working memory of the search kept in dynamic predicates local to each worker.

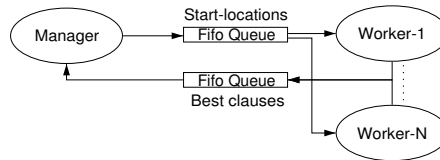


Figure 6.6: Concurrent Aleph. A manager schedules start points for a crew of workers. Each worker computes the best clause from the neighbourhood of the start point, delivers it to the manager and continues with the next start-point.

6.6.2.1 Experimental results and discussion

An exploratory study was performed to study the speedup resulting from using multiple threads on an SMP machine. We realised a work-crew model implementation for randomised local search in Aleph version 4. As the task is completely CPU bound we expected optimal results if the number of threads equals the number of utilised processors.⁵ The task consisted of 16 random restarts, each making 10 *moves* using the *carcinogenesis* (King and Srinivasan 1996) data set.⁶ This task was carried out using a work-crew of 1, 2, 4, 8 and 16 workers scheduled on an equal number of CPUs. Figure 6.7 shows that speedup is nearly optimal upto about 8 CPUs. Above 8, synchronisation overhead prevents further speedup. Note that later enhancements to messages queues as described in section 6.5.2 were not available for these tests.

The above uses one thread per CPU, the optimal scenario for purely CPU bound tasks. We also assessed the performance when using many threads per CPU using Aleph. These results indicate the penalty of converting a single-threaded design into a multi-threaded one.

⁵We forgot to reserve a CPU for the manager. As it has little work to do we do not expect results with an additional CPU for the manager to differ significantly from our results.

⁶<ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Datasets/carcinogenesis/progol/carcinogenesis.t>

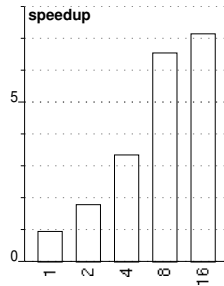


Figure 6.7: Speedup with an increasing number of CPUs defined as elapsed time using one CPU divided by elapsed time using N CPUs. The task consisted of 16 *restarts* of 10 *moves*. The values are averaged over 30 runs. The study was performed on a Sun Fire 6800 with 24 UltraSPARC III 900 MHz Processors, 48 GB of shared memory, utilising up to 16 processors.

Figure 6.8 shows the results. The X-axis show the number of used threads with the same meaning as used in the above experiment. The Y-axis shows both the (user) CPU time and the elapsed time. The top-two graphs show that on single CPU hardware there is no handicap upto 8 threads. With 16 and 32 threads, overhead starts to grow quickly. On dual-CPU hardware (bottom-two graphs), the situation is slightly different. The point for 1 thread (left) illustrate the difference in CPU speed between our single-CPU platform and SMP platform. With two threads, CPU time remains almost the same and elapsed time is reduced to almost 50%. As more threads are used, both CPU and elapsed time increase much faster than in the single-CPU scenario.

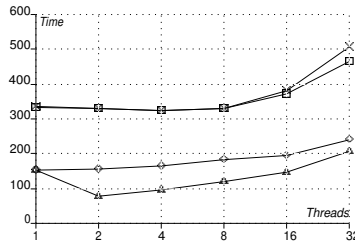


Figure 6.8: CPU- and elapsed time running Aleph concurrent on two architectures. The top two graphs are executed on a single CPU Intel PIII/733 Mhz, SuSE 8.2. The bottom two graphs are executed on a dual Athlon 1600+, SuSE 8.1. The X and triangle marked graphs represent elapsed time.

6.7 Related Work

This section provides an incomplete overview of other Prolog implementations providing multi-threading where threads only share the database. Many implementations use message queues (called *port* if the queue is an integral part of the thread or *channel* if they can be used by multiple threads).

SICStus-MT(Eskilson and Carlsson 1998) describes a prototype implementation of a multi-threaded version of SICStus Prolog based on the idea to have multiple Prolog engines only sharing the database. They used a proprietary preemptive scheduler for the prototype and therefore cannot support SMP hardware and have trouble with clean handling of blocking system-calls. The programmer interface is similar to ours, but they do not provide queues (channels) with multiple readers, nor additional synchronisation primitives.

CIAO Prolog⁷ (Carro and Hermenegildo 1999) provides preemptive threading based on POSIX threads. The referenced article also gives a good overview of concurrency approaches in Prolog and related languages. Their design objectives are similar, though they stress the ability to backtrack between threads as well as Linda-like (Carriero and Gelernter 1989) blackboard architectures. Threads that succeed non-deterministically can be restarted to produce an alternative solution and instead of queues they use ‘concurrent’ predicates where execution suspends if there is no alternative clause and is resumed after another thread asserts a new clause.

Qu-Prolog⁸ provides threads using its own scheduler. Thread creation is similar in nature to the interface described in this article. Thread communication is, like ours, based on exchanging terms through a queue attached to each thread. For atomic operations it provides `thread_atomic_goal/1` which freezes all threads. This operation is nearly impossible to realise on POSIX threads. Qu-Prolog supports `thread_signal/2` under the name `thread_push_goal/2`. For synchronisation it provides `thread_wait/1` to wait for arbitrary changes to the database.

Multi-Prolog(de Bosschere and Jacquet 1993) is logic programming instantiation of the Linda blackboard architecture. It adds primitives to ‘put’ and ‘get’ both passive Prolog literals and active Prolog atoms (threads) to the blackboard. It is beyond the scope of this article to discuss the merits of message queues vs. a blackboard.

6.8 Discussion and conclusions

We have demonstrated the feasibility of supporting preemptive multi-threading using portable POSIX thread primitives in an existing Prolog system developed for single-threading. Built on the POSIX thread API, the system has been confirmed to run unmodified on six Unix dialects. The MacOS X version requires special attention due to incomplete support for POSIX semaphores. The Windows version requires a fundamentally different implementation of atom garbage collection due to the lack of asynchronous signals. The

⁷<http://clip.dia.fi.upm.es/Software/Ciao/>

⁸<http://www.svrc.uq.edu.au/Software/QuPrologHome.html>

pthread-win32 emulation needed replacements using the native Windows API at various places to achieve good performance.

Concurrently running static Prolog code performs comparable to the single-threaded version and scales well on SMP hardware, provided that the threads need little synchronisation. Synchronisation issues appear in:

- *Dynamic predicates*
Dynamic predicates require mutex synchronisation on assert, retract, entry and exit. Heavy use of dynamic code can harm efficiency significantly. The current system associates a unique mutex to each dynamic predicate. Thread-local code only synchronises on entry to retrieve the clause list that belongs to the calling thread. In the future, we will consider using lock-free primitives (Gidenstam and Papatriantafilou 2007) for synchronising dynamic code, in particular thread-local code.
- *Atoms*
Creating an atom, creating a reference to an atom from `assert/1` or `recorda/1` as well as erasing records and clauses referencing atoms require locking the atom table. Worse, atom garbage collection affects all running threads, harming threads under tight real-time constraints. Multi-threaded applications may consider using SWI-Prolog's non-standard packed strings to represent text concisely without locking.
- *Meta-calling*
Meta-calling requires synchronised mapping from module and functor to predicate. The current system uses one mutex per module.
- *Passing messages over a queue*
Despite the enhancements described in section 6.5.2, passing messages between threads requires copying them twice and locking the queue by both the sender and receiver.

POSIX mutexes are stand-alone entities and thus not related to the data they protect through any formal mechanism. This also holds for our Prolog-level mutexes. Alternatively a lock could be attached to the object it protects (e.g., a dynamic predicate). We have not adopted this model as we regard the use of explicit mutex objects restricted to rare cases and the current model using stand-alone mutexes is more flexible.

The interface presented here does, except for the introduction of message queues, not abstract much from the POSIX primitives. Since the original publication of this paper we added higher level abstractions implemented as library predicates. The predicate `concurrent(N,Goals,Options)` runs M goals using N threads and stops if all work is done or a thread signalled failure or an error, while `first_solution(X,Goals,Options)` tries alternative strategies to find a value for X , stopping after the first solution. The latter is useful if there are different strategies to find an answer (e.g., literal ordering or breath vs. depth first search) and no way of knowing the best. We integrated threads into the development tools, providing source-level debugging for threads and a graphical monitor that displays status and resource utilisation of threads.

Manipulation of Prolog engines from foreign code (section 6.4.3) is used by the bundled Java interface (JPL) contributed by Paul Singleton as well as the C#⁹ interface contributed by Uwe Lesta.

Multi-thread support has proved essential in the development of Triple20 (chapter 2, Wielemaker et al. 2005) where it provides a responsive GUI application and ClioPatria (chapter 10, Wielemaker et al. 2008), where it provides scalability to the web server. Recently, we see a slowdown in the pace with which individual cores¹⁰ become faster. Instead, recent processors quickly accommodate more cores on one chip. This development demands support for multiple cores. The approach we presented allows for exploiting such hardware without much change to common practice in Prolog programming for the presented use-cases. In particular, Prolog hosted web-services profit maximally with almost no consequences to the programmer. Exploiting multiple cores for CPU-intensive tasks may require significant redesign of the algorithm. This situation is unfortunate, but shared with most today's programming languages.

Acknowledgements

SWI-Prolog is a Free Software project which, by nature, profits heavily from user feedback and participation. We would like to express our gratitude to Sergey Tikhonov for his courage to test and help debug early versions of the implementation. The work reported in section 6.6.2 was performed jointly with Ashwin Srinivasan and Steve Moyle at the Oxford University Computing Laboratory. We gratefully acknowledge the Oxford Supercomputing Centre for the use of their system, and in particular Fred Youhanaie for his patient guidance and support. Anjo Anjewierden has provided extensive comments on earlier drafts of this article.

⁹<http://gollem.science.uva.nl/twiki/pl/bin/view/Foreign/CSharpInterface35>

¹⁰The terminology that describes processing units has become confusing. For a long time, we had one chip that contained one processing unit, called CPU or processor. Now, there can be multiple 'cores' on a single physical chip that act as (nearly) independent classical CPUs. The term CPU has become ambiguous and can refer either to the chip or to a single core on a chip.