Native Preemptive Threads in SWI-Prolog

Jan Wielemaker

Social Science Informatics (SWI),
University of Amsterdam,
Roetersstraat 15, 1018 WB Amsterdam, The Netherlands,
jan@swi.psy.uva.nl

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 very close to standard Prolog while providing valuable extra functionality. This article describes the introduction of multiple threads in SWI-Prolog exploiting OS-native threading. 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.

1 Introduction

There are two approaches to concurrency in the Prolog community, implicit fine-grained parallelism where tasks share Prolog variables and systems (see Sect. 7) in which Prolog engines only share the database (clauses) and run otherwise completely independent. Programming these *multi-threaded* systems is very close to programming single-threaded Prolog systems, turning these systems into an attractive platform for tasks where concurrency is desirable:

Network servers/agents These systems must be able to pay attention to multiple clients. Threading allows multiple, generally almost independent, tasks to make progress at the same time and can improve overall performance when exploiting multiple CPUs (SMP) or if the tasks are I/O bound. Section 4.1 provides an example.

Embedding in multi-threaded servers Concurrent network-service infrastructures such as CORBA or .NET that embed Prolog can profit from multi-threaded Prolog retaining their overall concurrent behaviour, which is lost if request must be serialized to a single Prolog instance that is responsible for a significant part of the server's work.

Background processing in interactive systems Responsiveness and usefulness of interactive applications can be improved if background processing deals with tasks such as 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 6.2 describes an experiment.

In multi-threaded Prolog we must add primitives for threads to communicate and synchronise their activities. Our choices are based on the requirement to cooperate smoothly with multi-threaded foreign language code as well as the desire to keep it simple for the Prolog programmer.

Our implementation is based to the POSIX thread (pthread) API [2] for its portability and clean design. On Windows we use a mixture of pthread-win32[1] and the native Win32 thread-API.

This paper explores the loss of performance of single-threaded Prolog code executing in a multi-threaded environment. It also explores the consequences of introducing threads in an originally single-threaded implementation of the Prolog language including difficult areas such as atom garbage-collection.

In Sect. 2 we summary our requirements for multi-threaded Prolog. Next we describe what constitutes a thread and what primitives are used to make threads communicate and synchronise. In Sect. 4 we summarise our new primitives. Section 5 describes implementation experience, followed by performance analysis in Sect. 6 and an overview of related work in Sect. 7.

2 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.

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. Figure 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.

[1] http://sources.redhat.com/pthreads-win32/

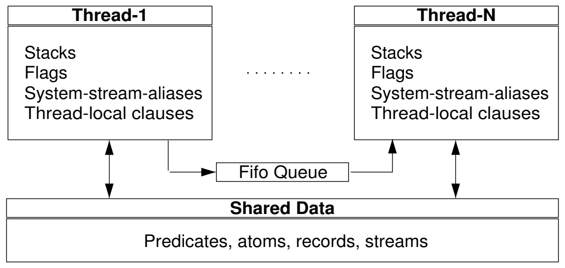


Fig. 1. Multiple Prolog engines sharing the same database. Flags and the system-defined stream aliases such as `current_input` are copied from the creating thread. Clauses are normally shared, except for thread-local clauses discussed below in Sect. 3.1.

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 Sect. 5. For dynamic predicates we kept the 'logical update semantics' as defined by the ISO standard [6]. 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 Sect. 3.2.

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*.

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. POSIX threads offer three mechanisms to organise this:

A mutex is a **Mut**ual **Ex**clusive 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 any 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 it 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 [2] explains in chapter 4, the commonly used thread cooperating techniques can be realised using the above two primitives. These primitives however are not very attractive to the Prolog user because great care is required to use them in the proper order and complete all steps of the protocol. Failure to do so may lead to data corruption or all threads waiting for an event to happen that never will (deadlock). Non-determinism, exceptions and the interactive development-cycle supported by Prolog complicate this further.

Our primary synchronisation primitive is a FIFO (first-in-first-out) queue of Prolog terms. This approach has been used successfully in similar projects and languages, see Sect. 7. 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 Sect. 4.2.

3.3 I/O and debugging

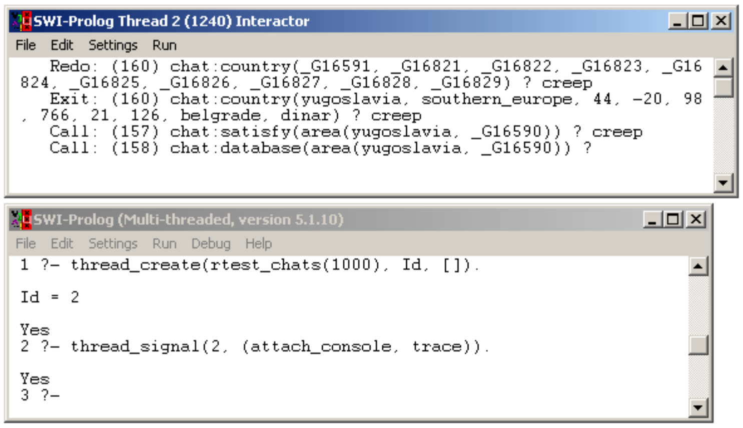
Support for multi-threaded I/O is rather primitive. I/O streams are global objects that may be created, accessed and closed from any thread knowing their handle. All I/O predicates lock a mutex associated with the stream, providing elementary consistency.

Stream alias names for the system streams (e.g. `user_input`) are thread-specific, where a new thread inherits the bindings from 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**, a primitive similar to **thread_push_goal/2** in Qu-Prolog and described in Sect. 4.2, the user can attach a console to any thread as well as start the debugger in any thread as illustrated in Fig. 2.

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.


SWI-Prolog Thread 2 (1240) Interactor
File Edit Settings Run

```
   Redo: (160) chat:country(_G16591, _G16821, _G16822, _G16823, _G16
824, _G16825, _G16826, _G16827, _G16828, _G16829) ? creep
   Exit: (160) chat:country(yugoslavia, southern_europe, 44, -20, 98
, 766, 21, 126, belgrade, dinar) ? creep
   Call: (157) chat:satisfy(area(yugoslavia, _G16590)) ? creep
   Call: (158) chat:database(area(yugoslavia, _G16590)) ?
```

SWI-Prolog (Multi-threaded, version 5.1.10)
File Edit Settings Run Debug Help

```
1 ?- thread_create(rtest_chats(1000), Id, []).

Id = 2

Yes
2 ?- thread_signal(2, (attach_console, trace)).

Yes
3 ?-
```

Fig. 2. Attach a console and start the debugger in another thread.

4.1 A short example

Before describing the details, we present the implementation of a simple network service in Fig. 3. We will not discuss the details of all built-in and library predicates used in this example. The thread-related predicates are discussed in more detail in Sect. 4.2 while all details can be found in [13]. 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.

4.2 Prolog primitives

This section discusses the built-in predicates we have added to Prolog. The description of the API is incomplete to keep it concise. A full description is in [13].

thread_create(*:Goal, -Id, +Options*)
Create a thread which starts executing *Goal*. *Id* is unified with the thread-

```
:- 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).
```

```
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', []).
```

Fig. 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.

identifier. The **thread_create/3** call returns immediately. *Goal* can succeed at most once.

The new Prolog engine runs independently. If the thread is *attached*, any thread can wait for its completion using **thread_join/2**. Otherwise all resources are reclaimed silently on completion.

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***)**
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.

thread_send_message(*+QueueOrThread, +Term***)**
Add a copy of term to the given queue or default queue of the thread. Return immediately.[2]

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.

[2] For a memory-efficient realisation of the pipeline model it may be desirable to suspend if the queue exceeds a certain length, waiting for the consumers to drain the queue.

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. 'First opportunity' is defined to be the next pass through the call-port or foreign code calling PL_handle_signals(). The latter mechanism is used to make threads handle signals during blocking I/O, etc. This primitive is intended for 'manager' threads to control their work-crew as illustrated in Fig. 4 and for the developer to abort or trace a thread (Fig. 2).

Worker	**Manager**

worker(Queue) :-
   thread_get_message(Queue, Work),
   catch(do_work(Work), **stop**, cleanup),
   worker(Queue).
	. . .
	thread_signal(Worker, **throw(stop)**),
	. . .

Fig. 4. Stopping a worker using **thread_signal/2**. Bold fragments show the relevant parts of the code.

4.3 Accessing Prolog threads from C

Integration with 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. The system provides two sets of primitives, one for long living external threads that want to use Prolog often and one to facilitate environments with many or short living threads that have to do some infrequent work in Prolog.

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 explicitely using PL_thread_destroy_engine() or it will be destroyed automatically when the underlying POSIX thread terminates. This method is not very suitable for many threads that infrequently require Prolog as creating and destroying Prolog engines is an expensive operation and engines require significant memory resources.

Alternatively, foreign code can create one or more Prolog engines using PL_create_engine() and attach an engine using PL_set_engine(). Setting 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.[3]

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 (Sect. 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 it as an extra variable to commonly used small functions as identified using the gprof [8] profiling tool. Mutex contention was analysed and reduced from some critical places:

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-reconsult cycle is left to a garbage collector that operates similarly to the atom garbage collection described in Sect. 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.

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

[3] http://gollem.swi.psy.uva.nl/twiki/pl/bin/view/Development/MultiThreadEmbed

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 *atom* mutex, preventing threads to create atoms or increment 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 [1]. During the execution of GC, it is very 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 a GC is running atom-GC is delayed until no thread executes GC.

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 Sect. 6.1. In Sect. 6.2 we studied the speedup on SMP systems by splitting a large task into subtasks that are distributed over a pool of threads.

6.1 Comparing multi-threaded to single threaded version

We used the benchmark suite by Fernando Pereira[4] for comparing the single threaded to the multi threaded version on a range of benchmarks addressing very specific parts of the Prolog implementation. We normalised the iterations of each test to make it run for approx. one second, after which we executed the 34 tests in 5 different settings, described here in the left-to-right order used in Fig. 5. All test were run on a 550Mhz Crusoe machine running SuSE Linux 7.3 and Windows 2000.

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

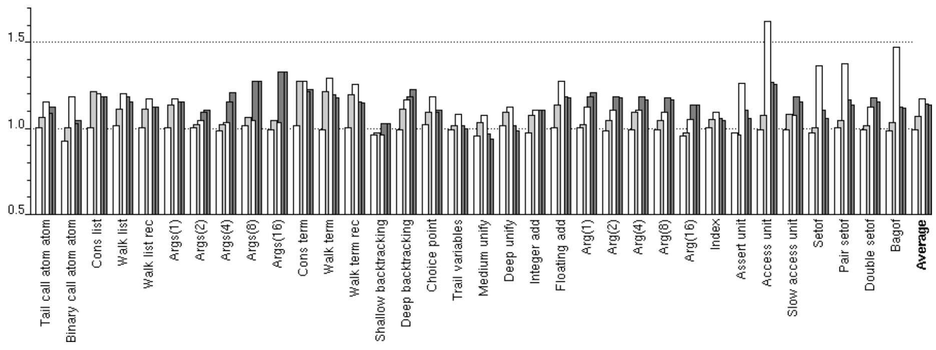


Fig. 5. Performance comparison between single and multi-threaded versions. The Y-axis shows the time to complete the benchmark in seconds.

Figure 5 indicates there is no significant difference on any of the tests between the single- and multi-threaded version on Windows 2000. It does show significant differences on Linux, where the single-threaded version is considerably faster and the multi-threaded version performs overall slightly worse and a few tests that perform much less. The poorly performing tests all require frequent mutex synchronisation due to the use of dynamic predicates or, for tests using **setof/3**, locking atoms in the result table.

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 version that passes a pointer to the virtual machine state but is otherwise identical to the single threaded version. This version (2nd bar)

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

exhibits behaviour very 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 and the difference does not show up in the Windows version due to inferior optimisation of MSVC 5 compared to gcc 2.95.[5] We also conclude that Microsoft *critical sections* used in the Windows version are considerably faster than the glibc implementation of the more general POSIX mutex objects.

Finally we give the cumulative results of 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.

6.2 A case study: Speedup on SMP systems

This section describes the results of multi-threading the Inductive Logic Programming system Aleph [11], 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) ($\bot$) [10].

Many strategies are possible for searching this often huge lattice. Randomised local search [12] 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 Fig. 6. We exploited the thread-local predicates described Sect. 3.1 to make the working memory of the search kept in dynamic predicates local to each worker.

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

[5] This could be verified by compiling Prolog using GCC on Windows. This test has not been performed.

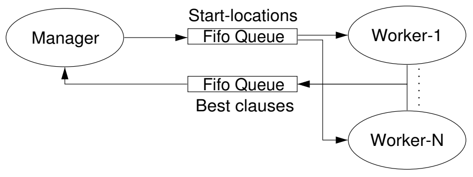


Fig. 6. Concurrent Aleph. A manager schedules start points for a crew of workers. Each worker computes the best clause from the neighbourhood of the start point, delivers it to the manager and continues with the next start-point.

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.[6] The task consisted of 16 random restarts, each making 10 *moves* using the *carcinogenesis* [9] data set.[7] 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 7 shows the result.

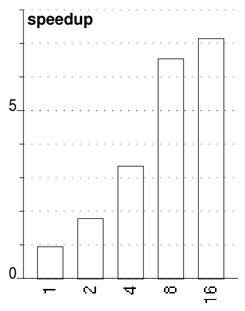


Fig. 7. Speedup with an increasing number of CPUs defined as elapsed time using one CPU divided by elapsed time using N CPUs. The task consisted of 16 *restarts* of 10 *moves*. The values are averaged over 30 runs. The study was performed on a Sun Fire 6800 with 24 UltraSPARC III 900 MHz Processors, 48 GB of shared memory, utilising up to 16 processors. Each processor had 2 GB of memory.

Finally, we used Aleph to assess performance using many threads per CPU. These results indicate the penalty of splitting a single-threaded design into a multi-threaded one. The results are shown in Fig. 8.

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

[7] ftp://ftp.comlab.ox.ac.uk/pub/Packages/ILP/Datasets/carcinogenesis/progol/carcinogenesis.tar.Z

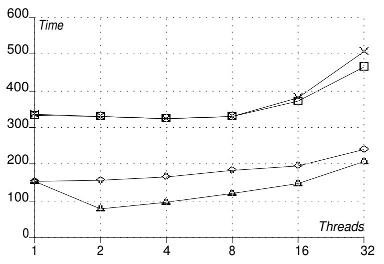


Fig. 8. CPU- and elapsed time running Aleph concurrent on two architectures. The top two graphs are executed on a single CPU Intel PIII/733 Mhz, SuSE 8.2. The bottom two graphs are executed on a dual Athlon 1600+, SuSE 8.1. The X and triangle marked graps represent elapsed time.

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 [7] 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 Prolog [8] [4] 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 [3] 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-Prolog [9] 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.

[8] http://clip.dia.fi.upm.es/Software/Ciao/
[9] http://www.svrc.uq.edu.au/Software/QuPrologHome.html

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 [5] 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.

8 Issues and future work

Concurrency running static Prolog code is very good. Performance however degrades quickly on SMP systems when using primitives that require synchronisation. The most important issues are:

Dynamic predicates harm concurrency Dynamic predicates require mutex synchronisation on assert, retract, entry and exit. Heavy use of dynamic code can harm efficiency significantly. Thread-local dynamic code can avoid expensive synchronisation.

Atoms harm concurrency Atom handling in the current implementation has serious flaws. 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. Even worse, atom garbage collection affects all running threads, harming threads under tight real-time constraints.

Meta-calling harms concurrency Meta-calling requires synchronised mapping from module and functor to predicate.

Mutex synchronisation 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 (i.e. a dynamic predicate). We have not adopted this model as we regard the use of explicit mutex objects restricted to special cases.

9 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. Concurrently running static Prolog code performs comparable to the single-threaded version and scales well on SMP hardware. Threaded Prolog using a shared database is relatively easy to implement while providing valuable functionality for server, interactive and CPU-intensive applications.

Built on the POSIX thread API, the system has been confirmed to run unmodified on six Unix dialects. The MacOS X and MS-Windows versions required special attention due to the partial support for POSIX semaphores in MacOS X and the lack of asynchronous signals in MS-Windows.

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 Sect. 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 versions of this article.

References

1. Karen Appleby, Mats Carlsson, Seif Haridi, and Dan Sahlin. Garbage collection for Prolog based on WAM. *Communications of the ACM*, 31(6):719–741, 1988.
2. David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley, Reading, MA, USA, 1997.
3. Nicholas Carriero and David Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, April 1989.
4. Manuel Carro and Manuel V. Hermenegildo. Concurrency in Prolog using threads and a shared database. In *International Conference on Logic Programming*, pages 320–334, 1999.
5. Koen de Bosschere and Jean-Marie Jacquet. Multi-Prolog: Definition, operational semantics and implementation. In David S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*, pages 299–313, Budapest, Hungary, 1993. The MIT Press.
6. P. Deransart, A. Ed-Dbali, and L. Cervoni. *Prolog: The Standard.* Springer-Verlag, New York, 1996.
7. Jesper Eskilson and Mats Carlsson. SICStus MT—a multithreaded execution environment for SICStus Prolog. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Programming Languages: Implementations, Logics, and Programs*, volume 1490 of *Lecture Notes in Computer Science*, pages 36–53. Springer-Verlag, 1998.
8. Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.
9. R.D. King and A. Srinivasan. Prediction of rodent carcinogenicity bioassays from molecular structure using inductive logic programming. *Environmental Health Perspectives*, 104(5):1031–1040, 1996.
10. S. Muggleton. Inverse entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming*, 13(3-4):245–286, 1995.
11. A. Srinivasan. *The Aleph Manual*, 2003.
12. F. Železný, A. Srinivasan, and D. Page. Lattice-search runtime distributions may be heavy-tailed. In S. Matwin and C. Sammut, editors, *Proceedings of the 12th International Conference on Inductive Logic Programming*, volume 2583 of *Lecture Notes in Artificial Intelligence*, pages 333–345. Springer-Verlag, 2003.
13. J. Wielemaker. *SWI-Prolog 5.1: Reference Manual.* SWI, University of Amsterdam, Roetersstraat 15, 1018 WB Amsterdam, The Netherlands, 1997-2003. E-mail: jan@swi.psy.uva.nl.