

# Shared-Memory Multiprocessing for Interactive Theorem Proving

Makarius Wenzel

Université Paris-Sud 11, LRI  
Orsay, France  
<http://www.lri.fr/~wenzel/>

**Abstract.** We address the multicore problem for interactive theorem proving, in particular for the Isabelle platform where substantial support for parallel theory and proof processing has emerged in the past few years. This achieves pervasive parallelism without user intervention and gains significant speedup factors.

Ever since the stagnation of clock frequency in 2005, hardware manufacturers have imposed the burden of explicit parallelism to application developers. Now the cumulative results of decades of research in parallel programming need to be revisited and turned into concrete performance figures for each kind of application separately. Interactive theorem proving with its emphasis on explicit proof checking provides particularly good starting conditions for pervasive parallelization, although it turns out that many aspects of the system have to be reworked in subtle ways. This includes the implementation language, the inference kernel, and the processing of theory and proof commands. We discuss common requirements, problems, and solutions according to the current state of Isabelle2011. Concrete performance figures are analyzed for recent commodity hardware at the transition from 8 to 16 cores.

## 1 Motivation

### 1.1 The Multicore Problem

Software developers have become accustomed to *Moore's Law* of computing hardware, which states that chip density and integrated functionality doubles approximately every two years. In the past, this was correlated with an increase in clock frequency, so that existing programs would become exponentially faster over time, or the complexity of programs could be increased at that rate without the user noticing too much about “software bloat”.

Around 2005 the rules have changed dramatically, when clock frequency has reached a plateau of 3.0 GHz, mainly due to excessive power dissipation and overheating at higher frequencies. Chip manufacturers have proclaimed that exponential growth will continue only in the number of explicitly visible CPU cores. So personal computers with 2–32 hardware threads have become available.

This naturally poses challenges to application software development, as is emphasized by the wakeup-call “The Free Lunch is Over” [17]. Sequential code that

fails to adapt to this evolutionary pressure will suffer from exponential decline of relative performance, as the number of CPU cores multiply. Joe Armstrong illustrates this problem as follows, when presenting concepts of “concurrency-oriented programming” in Erlang [2]:

number of cores	max. CPU usage	Joe Armstrong says:
2	50 %	“2 cores won’t hurt you”
4	25 %	“4 cores will hurt a little”
8	12 %	“8 cores will hurt a bit”
16	6 %	“16 cores will start hurting”
32	3 %	“32 cores will hurt a lot”

We refrain from speculating what happens beyond 32 cores, where sequential computing power quickly converges to 0%. This will require another shift of paradigm beyond the present multicore challenge.

An important characteristic of the range of 2–32 cores is that hardware manufacturers can still uphold the convenient programming model of *shared memory* between parallel threads (in the same process address space), with reasonable memory bandwidth for transfers between CPU modules. This requires an increasingly complex memory hierarchy of caches and quick paths in the memory subsystem, but it is one of the points where the sustained exponential increase of hardware capabilities is still invested after the stagnation of clock frequency.

Subsequently we assume this hardware class with 2–32 general-purpose CPU cores and 2–32 GB of shared main memory, and the simplifying model of uniform memory access between all cores. For the concrete measurements, we shall use a 3rd generation Mac Pro (early 2009) with 8 CPU cores and 16 hardware threads (2 × 4-core hyperthreading Intel Xeon at 2.93 GHz) and 32 GB main memory (DDR3 at 1066 MHz), running Mac OS X Snow Leopard in genuine 64 bit mode.

## 1.2 LCF-style Interactive Theorem Proving

Due to various characteristics, interactive theorem provers like HOL [23, §1], Coq [23, §4], Isabelle [23, §6], or ACL2 [23, §8] fit well into the above model of shared-memory multiprocessing.

**Functional programming with mainly immutable data.** Typical provers are implemented in a higher-order functional programming language (LISP, ML, Haskell) with a strong emphasis on large symbolic data structures that are immutable (e.g. big  $\lambda$ -terms for syntax or proof terms). In the multicore era, immutability is one of the inherent advantages of pure functional programming, and even mainstream Java programmers have noticed that (cf. the attention that functional-object-oriented Scala [15] and the LISP dialect Clojure have gained in the JVM world). Shared memory allows to pass pointers to immutable data without requiring copying, and without any danger of data corruption between application threads. Moreover, structural equality

of pure values (as defined in Standard ML and Haskell) enables the runtime system to produce distributed copies without special precautions about coherence between different processors; e.g. see the parallel garbage collector of Glasgow Haskell [11]. Low-level hardware caches (“store buffers”) also perform better, when results published to global memory are never modified again after local initialization.

Thus ML and Haskell programs can afford thread-based parallelism, without the hazards known from C/C++ or FORTRAN. Nonetheless, threads and synchronization primitives are difficult to use directly in application code, so higher principles of parallel functional programming will be required.

**LCF-style abstraction of formally certified entities.** In the original LCF architecture [5] that is observed by the HOL family and Isabelle, certified entities like theorems, well-formed terms, or background theory certificates [21] are represented as entities of abstract datatypes. Thus the corresponding “proof objects” only exist as Platonistic idea, without direct representation in memory. Such abstract datatype values can be easily transferred in shared memory by the runtime system, with the same type-safety properties as the original ML design [4]. In contrast, explicit communication of results between separate process address spaces would require some means to externalize formal entities. Depending how thoroughly proof checking is treated at the kernel level, this may also demand full proof terms to be communicated, say over a network of CPU nodes.

**Continuous interaction with a large prover process.** The typical prover interaction model is centered around a single process with a relatively large context of background theories, where the user incrementally produces relatively small additions that are associated with short running executions (tactic applications etc.). This scenario can be efficiently represented by a single multi-threaded process. Separate processes would introduce various complications here due to the following observations.

Startup time of a fresh prover process can be very high (several seconds in Isabelle), but the expected latency for incremental user operations is very low (in the range of milliseconds). Instead of running threads within the same process, one could try a Unix-style fork of the whole process with the usual “copy-on-write” strategy for virtual memory space. Unfortunately, after some rounds of garbage collection sharing of semantically immutable data is lost, because the runtime system has moved it in memory.

This should make sufficiently clear that interactive theorem proving and shared-memory multiprocessing are worth investigating more thoroughly. We shall provide a general overview of many questions that arise when embarking on such a project, and provide some clues how the answers of Isabelle (and the underlying Poly/ML platform) could be transferred to other provers, which are all still mostly sequential.

After decades of research into parallel programming, there is occasionally a tendency to treat the topic as settled, although there are few general answers. Universal compilation schemes from sequential algorithms to parallel code do

not exist, and each kind of application needs to be investigated separately for parallelization strategies. In doing this for interactive theorem proving, our main goal is to provide *implicit parallelism* for user developments within the prover. This means the prover infrastructure takes care of the details of organizing theory and proof checking efficiently on multicore hardware.

The rest of the paper is structured as follows. §2 explores some parallelization strategies by concrete experiments with Isabelle2011 and the Archive of Formal Proofs. §3 introduces the main aspects of our parallel prover architecture, which covers the underlying functional programming language, the LCF-style inference kernel, and goal-oriented proof construction. §4 provides further performance analysis of the current parallelization strategies in Isabelle.

## 2 Strategies for Parallel Proof Checking

Subsequently, our running example is the medium-sized entry from *The Archive of Formal Proof* called *Jinja-Slicing* (<http://afp.sf.net/entries/Slicing.shtml>), which takes about 788 s (13 min) when checked sequentially on a single core.

From the empirical evidence over several years, the typical wall-clock runtime for the re-checking of finished formalizations is in the range of 10 min to 100 min. Thus such absolute figures are roughly correlated with the size and complexity or formal theory developments that are feasible with the interactive prover. Any speedup gained here will eventually translate into larger formalizations.

### 2.1 Peep-hole Parallelism

Inspecting the sources of *Jinja-Slicing* reveals the following situation at line 1167 of `Slicing/JinjaVM/JVMCFG_wf.thy`:

```
(* This takes veeery long! *)  
by simp_all
```

At that point a goal state of 1225 subgoals is solved by simplification in 80 s elapsed time. Since all subgoals happen to be independent (without any instantiable variables that could influence each other) this is an isolated case of an “embarrassingly parallel” problem. The idea is to simplify the subgoals independently (via a parallel version of `List.map` in ML) and recombine the results by back-chaining with the original goal state. (Every prover should have a way to express that, e.g. via auxiliary implications.)

The above Isabelle/Isar proof method invocation corresponds to the tactic `ALLGOALS (asm_full_simp_tac @{simpset})` so this can be parallelized by the alternative `PARALLEL_GOALS` tactical that is already available in Isabelle:

```
by (tactic "PARALLEL_GOALS (asm_full_simp_tac @{simpset} 1)")
```

Of course, there is a small overhead to be invested for management of parallel evaluations and the logical decomposition / recombination of the goal state. So 16 cores will hardly translate into factor 16 speedup. Concrete timing results for this experiment are as follows:

worker threads	elapsed time	CPU time	pseudo speedup	real speedup
$m$	$\varepsilon(m)$	$\zeta(m)$	$\zeta(m) / \varepsilon(m)$	$\varepsilon(1) / \varepsilon(m)$
1	79.7 s	79.7 s	1.0	1.0
2	39.9 s	79.1 s	2.0	2.0
4	20.1 s	81.2 s	4.0	4.0
8	11.2 s	82.9 s	7.4	7.1
12	9.7 s	102.9 s	10.6	8.2
16	8.3 s	116.4 s	14.0	9.6

This looks promising: we have managed to reduce the elapsed runtime from 80 s to 8 s on 8 cores with hyperthreading (16 worker threads), gaining a factor of 10. Does that mean we have solved the multicore problem? Not yet. We have achieved little more than an isolated boost of performance (which might still be useful in interactive development).

Empirical results of parallel performance always need to be treated with great care. The figures of elapsed time vs. CPU times are based on standard timers of the operating system, which we take for granted here. Presenting the ratio in the column “pseudo speedup” prominently to end-users is very tempting, since it is easy to produce on the spot for the current execution, and it gives the user a good feeling that his parallel hardware is actually being used by the application (albeit CPU cycles could have been wasted). In contrast, the “real speedup”  $\varepsilon(1) / \varepsilon(m)$  represents the success of parallelization more faithfully, although it is less exciting in the presentation and  $\varepsilon(1)$  is often unknown in practice.

Nonetheless, our result of 9.6 for 16 threads is still sufficiently close to the ideal of linear speedup, for a machine that has only 8 proper cores.

Further critical investigation of the experimental setup helps to assess the results of this typical “micro benchmarks”.

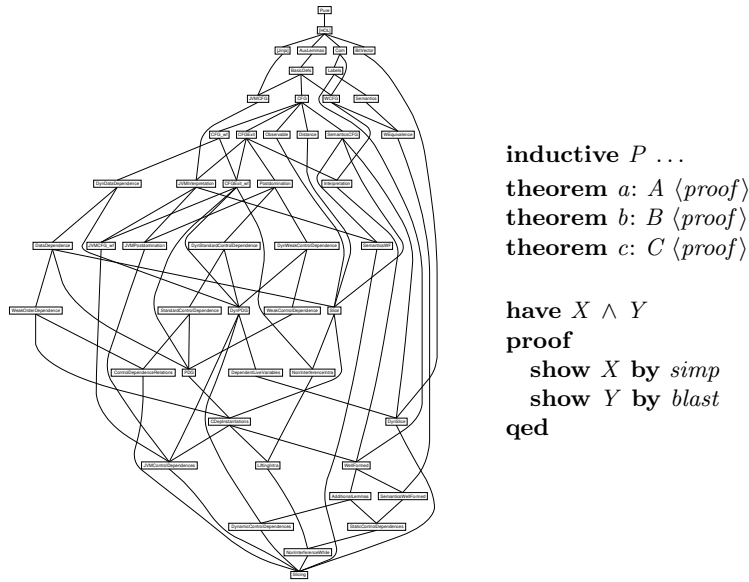
1. The total amount of CPU time consumed here is too low to heat the hardware significantly, so it runs at higher clock frequency than usual for 8–16 cores working on big problems for longer time. Sustained performance figures are expected to be lower than factor 10.
2. Very short runtime like 8 s ignores the amortized costs produced by heap allocations. Garbage collection at that point can easily add 2 s and spoil the large factor. GC is a general bottle-neck for shared-memory parallelization. Even if GC is itself parallel, it naturally requires high memory bandwidth and thus gains less from parallelization than regular application code.
3. Reducing 80 s to 8 s (factor 10) shortens the overall runtime of *Jinja-Slicing* from 788 s to 708 s (factor 1.1), which means we have gained only 10 % total performance improvement on 16 cores for far. It is unclear whether there are more such immediately parallel tactic applications in the formalization.

Points (1) and (2) certainly involve difficult problems that need to be addressed when building parallel hardware and parallel programming language runtime systems, respectively. Point (3) is more embarrassing, because it affects the internal organization of our own application, which often turns out to be the weakest link in the chain.

*Amdahl's Law* estimates the sub-linear speedup as  $1 / (s + p/m)$ , where  $s$  is the part of the program running sequentially, and  $p$  the part running in parallel (normalized such that  $s + p = 1$ ). For  $m \rightarrow \infty$  this converges to  $1 / s$ . In other words, the overall success of parallelization depends on the remaining fraction of inherently sequential code. This prediction can be made even more pessimistic by including the losses caused by organization of parallel computation, so that for very large number of cores the speedup would approach 0.

## 2.2 Pervasive Theory and Proof Parallelization

The main lesson learned from the previous experiment is that parallelism needs to be *pervasive* to gain significant speedup, i.e. the remaining sequential portion of the application runtime needs to become  $\approx 0$ . On order to get anywhere close to that we need to investigate our problem structure more thoroughly. We shall do that at different levels of granularity, as specified by parameter  $q$  below.



**Fig. 1.** Typical theory dependencies and rough content structure.

- Granularity  $q = 0$ : parallel theories.  
 Typical formalizations consist of an acyclic graph of theory nodes that depend on each other, often with a reasonably degree of independent paths, e.g. see figure 1 (left). Traversing this graph in depth-first order and composing nodes in a bottom-up manner, we gain some potential for parallelism in correlation with the breadth of the graph and the runtime for each node. This is similar to `make -j` on the command-line, but here we run multiple threads within the same ML process.
- Granularity  $q = 1$ : parallel theories and toplevel proofs.  
 As sketched in figure 1 (right), each theory mainly consists of a sequence of definition–statement–proof, with *specifications* of results given in the text and proofs that are *irrelevant* in practice.<sup>1</sup> Also note that definitions occasionally require proofs as justification internally.  
 Even though proofs are hard to produce and take very long to check, they are not required to process the outermost sequence of specifications. Proofs can be forked immediately and need to be joined only in the very end, when the whole graph of loaded theories is consolidated.
- Granularity  $q = 2$ : parallel theories, toplevel proofs, and end-proofs in Isar.  
 Further inspection of the Isar proof language [18] reveals extra potential for parallel checking. The high degree of compositionality in the language design admits to decompose proof checking systematically. Here we do this in the isolated situation of end-proofs “*by method*”. This works for proper Isar proof outlines, because most of the time for checking is spent at terminal positions, where claims emerging from top-down decomposition are finally established by arbitrary proof tools (*simp*, *blast*, *auto*, *force* etc.).

Here are some concrete results for these parallelization strategies:

worker threads	real speedup	real speedup	real speedup
$m$	$q = 0$	$q = 1$	$q = 2$
1	1.0	1.0	1.0
2	1.4	1.8	1.8
4	1.7	2.4	3.5
8	1.7	2.6	3.9

The full combination of  $q = 2$  achieves fairly good speedup of 3.5 on 4 cores, but the remaining imperfection becomes apparent when scaling further towards 8 cores. This again indicates some remaining sequentiality in the application.

Closer inspection of *Jinja-Slicing* reveals that there are several big tactic scripts inside large Isar proofs that are treated as single chunks. So  $q = 2$  is not sufficient to scale further in this case of weakly structured proofs. For further refinement, granularity  $q = 3$  could mean full parallelization of nested proof

<sup>1</sup> In classical LCF-style systems like HOL or Isabelle proof-irrelevance is a formal property of the basic calculus, but in rare situations extra-logical code might retrieve information about inferences. In contrast, Coq proof terms can formally occur in other terms, but practical developments mostly work with “opaque” proof terms.

texts, such that flat scripts inside larger proofs would also benefit, but it will require further reworking of Isar proof interpretation [18].

As we shall see in §4, better-structured Isabelle applications can scale into the range of 8 cores or more with  $q = 2$  already.

Note again that anything beyond parallelization of theory graphs and toplevel proofs is based on specific properties of the structured Isar proof language. Our data indicates that the generic scheme  $q = 1$  for average formalizations only scales up to 4 cores. For other provers, the particular situation will have to be inspected separately, in order to see how the 8 core category can be approached.

### 3 Parallel Prover Architecture

#### 3.1 Value-oriented Parallelism in ML

To make the computing power of shared-memory multi-processing available to the prover, we need to find suitable abstractions that achieve a good balance of flexibility and simplicity. Explicit threads and synchronization is too tedious and error-prone to work with. Note that system-level threads cannot be forked in arbitrary numbers without degrading performance; it is best to use a small number of worker threads corresponding to the available hardware execution units. Large numbers of small-scale ML evaluations need to be managed within a separate task queue that feeds a pool of worker threads.

In functional programming it is natural to organize parallel executions in a data-oriented manner. The concept of *future value* goes back to Multiplisp [7] at the least. The idea is to represent the eventual result with its associated evaluation task as first-class entity. An attempt to retrieve an unfinished result synchronizes with the evaluation process, hiding the full complexity of waiting and signalling between threads.

Our version of futures in Isabelle/ML is implemented on top of the raw multi-threading in Poly/ML [12], which is based on the well-known Posix Threads library. We carefully observe the Standard ML semantics, with its strict functional evaluation, possible synchronous exceptions, and asynchronous interrupts (according to SML'90). The main operations are as follows:

```
type  $\alpha$  future
val Future.fork: ( $unit \rightarrow \alpha$ )  $\rightarrow$   $\alpha$  future
val Future.join:  $\alpha$  future  $\rightarrow$   $\alpha$ 
val Future.cancel:  $\alpha$  future  $\rightarrow$  unit
```

The key observation is that *Future.join* (*Future.fork* (**fn** ()  $\Rightarrow$  *expr*)) produces the same result as *expr* outright, but the fork can be separated from the join to enable parallel evaluation. The overhead for fork + join is about  $10^{-6} \dots 10^{-5}$  s and the system can easily cope with  $10^5 \dots 10^6$  pending tasks.

Note that unfinished futures that are unwanted need to be canceled explicitly: there is no disposal of tasks of futures that happen to be inaccessible. Finished futures are subject to regular garbage collection. Exceptions raised by evaluating

*expr* are propagated to the join point. E.g. `val x = Future.fork (fn () => raise Match)` succeeds, but any later `Future.join x` raises exception `Match`.

The following example of `parallel_map: (α → β) → α list → β list` illustrates how futures can be aggregated to implement combinators that express “skeletons” for parallel programs:

```
fun parallel_map f xs =
  map Future.join (map (fn x => Future.fork (fn () => f x)) xs)
```

Parallel combinators like `map` (or the renowned `reduce`) are convenient, but our proof parallelization scheme uses the more flexible `Future.fork` and `Future.join` directly, because proofs emerge dynamically during the exploration of unknown theory content. Since forks are under program control, we can easily restrain substructural parallelization tasks (e.g. nested proofs) according to certain tuning parameters. This helps to avoid parallelization overhead in situations where it is likely that the available cores remain saturated for quite some time.

### 3.2 Promised Proofs

The core inference system of Isabelle is based on Natural Deduction, deriving judgments of the form  $\{A_1, \dots, A_n\} \vdash B$  with rules for  $\implies$  and  $\wedge$  that move propositions between the antecedent  $\Gamma$  and the succedent  $B$ . Subsequently, we include explicit proof objects as  $\lambda$ -terms, although they might be omitted in the implementation. So  $p : B$  means that  $p$  is a proof term for proposition  $B$ .

Inferences are always relative to a background theory  $\Theta$ , which contains constants for types, terms, and proofs (axioms), and auxiliary user data [22].

Proof holes work like global axioms, but need to be managed separately so that they can be fulfilled later (by proof substitution). Subsequently we introduce a pro-forma collection  $\Pi$  of promised proofs, and pass it through monotonically. Thus we can describe the deductive system as a judgement  $\Theta, \Pi, \Gamma \vdash p : B$  that is defined inductively by the following rules:

$$\frac{\Theta, \Pi, \Gamma \vdash q : B}{\Theta, \Pi, \Gamma - \{p : A\} \vdash (\lambda p : A. q) : (A \implies B)} \text{ (imp\_intro)}$$

$$\frac{\Theta_1, \Pi_1, \Gamma_1 \vdash p : (A \implies B) \quad \Theta_2, \Pi_2, \Gamma_2 \vdash q : A}{\Theta_1 \cup \Theta_2, \Pi_1 \cup \Pi_2, \Gamma_1 \cup \Gamma_2 \vdash p q : B} \text{ (imp\_elim)}$$

$$\frac{\Theta, \Pi, \Gamma \vdash p[x] : B[x] \quad x \notin \text{FV } \Gamma}{\Theta, \Pi, \Gamma \vdash (\lambda x. p[x]) : (\wedge x. B[x])} \text{ (all\_intro)}$$

$$\frac{\Theta, \Pi, \Gamma \vdash p : (\wedge x. B[x])}{\Theta, \Pi, \Gamma \vdash p a : B[a]} \text{ (all\_elim)}$$

$$\frac{}{\Theta, \Pi, \{p : A\} \vdash p : A} \text{ (assm)}$$

$$\begin{array}{c}
\frac{(c : A[? \bar{\alpha}]) \in \Theta}{\Theta, \emptyset, \emptyset \vdash c : A[\bar{\tau}]} \text{ (axiom)} \\
\frac{\Theta, \Pi, \Gamma \vdash p[\alpha] : B[\alpha] \quad \alpha \notin \text{TV } \Gamma}{\Theta, \Pi, \Gamma \vdash p[? \alpha] : B[? \alpha]} \text{ (type\_gen)} \\
\frac{\Theta, \Pi, \Gamma \vdash p[? \alpha] : B[? \alpha]}{\Theta, \Pi, \Gamma \vdash p[\tau] : B[\tau]} \text{ (type\_inst)}
\end{array}$$

Setting  $\Pi = \emptyset$  we get the existing inference system of Isabelle/Pure [21], which we shall use as a reference point for logical correctness.

Rules *type\_gen* and *type\_inst* require special attention. This is how Isabelle (and the HOL family in general) simulate outermost type quantification in the style of ML polymorphism. In Isabelle notation,  $\alpha$  refers to a *free* and  $? \alpha$  to a *schematic* type variable; this provides a syntactic hint if a type is considered locally fixed or arbitrary. The above type manipulations essentially emerge as admissible rules (by induction over derivations) as follows: the base case works because the axiomatic basis of  $\Theta$  is closed by arbitrary type substitutions; the step case works by pushing substitutions through the other inferences.

This naive polymorphism introduces an extra twist in maintaining promised proofs, because we need to track type substitutions and replay them when proofs are fulfilled eventually. If we would not care about schematic polymorphism, deferred proofs could be simulated outside the kernel as assumptions [1].

A *proof promise* is a constant of the form  $a[? \bar{\alpha}] : A[? \bar{\alpha}]$  where  $a$  is some identifier and  $A$  is a closed proposition containing only schematic type variables  $? \bar{\alpha} = ? \alpha_1, \dots, ? \alpha_n$  (in canonical order). Promises shall be treated like axioms with an explicit indication of type instances; rules *type\_gen* or *type\_inst* will produce concrete instances  $a[\bar{\tau}]$  at different occurrences in a proof term. Moreover, we define proof substitution  $p[a := q]$  for a closed proof  $q : A$  (where  $\text{TV } q = ? \bar{\alpha}$ ):

$$\begin{array}{l}
(a[\bar{\tau}] : A[\bar{\tau}])[a := q] = q[\bar{\tau}] \\
(b[\bar{\tau}] : B[\bar{\tau}])[a := q] = (b[\bar{\tau}] : B[\bar{\tau}]) \quad \text{if } a \neq b \\
(c : A)[a := q] = (c : A) \\
(p_1 p_2)[a := q] = (p_1[a := q]) (p_2[a := q]) \\
(\lambda x. p)[a := q] = \lambda x. p[a := q]
\end{array}$$

In other words,  $q$  is treated as definition for  $a$  that is expanded throughout the proof term, while propagating type substitutions accordingly. This is analogous to polymorphic *let* expressions within proof terms. It is particularly important to note here that substituting a closed term does not interfere with  $\lambda$ -abstraction.

The parallel Isabelle inference system is now defined inductively by the previous rules together with these extra ones:

$$\begin{array}{c}
\frac{\text{FV } A = \emptyset \quad \text{TV } A = \{? \bar{\alpha}\}}{\Theta, \{a : A\}, \emptyset \vdash a[? \bar{\alpha}] : A[? \bar{\alpha}]} \text{ (promise)} \\
\frac{\Theta_1, \Pi, \Gamma \vdash p : B \quad \Theta_2, \emptyset, \emptyset \vdash q : A \quad \Theta_2 \subseteq \Theta_1}{\Theta_1, \Pi - \{a : A\}, \Gamma \vdash p[a := q] : B} \text{ (fulfill)}
\end{array}$$

This models the idea of introducing and eliminating locally defined proof terms, without any particular policy how this is organized computationally. Thus parallelism is only implicitly present: *promise* produces a slot for a future certification of  $a : A$ , and *fulfill* joins derivation of  $p : B$  that depends on  $a : A$  with an actual  $q : A$  that is produced independently. For simplicity, the fulfilled result is required to be closed, with promises  $\Pi_2 = \emptyset$  and premises  $\Gamma_2 = \emptyset$ .

The additional requirement  $\Theta_2 \subseteq \Theta_1$  ensures that the resulting theory of the eventual result is indeed unchanged:  $\Theta_1 \cup \Theta_2 = \Theta_1$ . Otherwise one could invent circular proofs depending on later theorems, for example. The Isabelle kernel already supports a notion of certificates for background theory contexts [22, 21], with efficient operations to maintain monotonicity and check inclusions such as  $\Theta_2 \subseteq \Theta_1$ . Older variants of the HOL family would have to re-invent ways to work simultaneously with theorems in different background contexts.

### 3.3 Future theorems

In order to integrate proof terms with promises into the LCF-style inference kernel, the internal representation of type *thm* is augmented by an environment of pairs  $(a, q)$  where  $a$  is a promise identifier and  $q$  a future proof. Whenever a promise  $a : A$  is created, its eventual fulfillment has to be given as well, as a future  $q : A$  that can be produced in parallel (§3.1). Thus we get the following additional operations of the inference kernel:

```

val Thm.future: thm future  $\rightarrow$  term  $\rightarrow$  thm
val Thm.join_proofs: thm list  $\rightarrow$  unit

```

For example, *Thm.future* (*Future.fork* (**fn**  $() \Rightarrow$  *prf*))  $A$  produces a theorem of proposition  $A$ , but the actual proof is delivered later by evaluating the given *prf*: *thm*. The kernel ensures that the result really fits the original specification  $A$  and checks the required theory inclusion of the *fulfill* inference (§3.2).

Future theorems contain promised proofs as holes. By using unfinished results in other proofs, pending promises accumulate monotonically. *Thm.join\_proofs* ensures that the loose ends of given theorems are really closed. This time-consuming operation is performed by the theory loader at the very end, for all theorems that are reachable by the user within the theory name space. Any failures in proof promises will emerge at that point.

In the spirit of the original LCF architecture, the above inference kernel extension is quite minimalistic, it is limited to closed statements. When working relatively to some local proof context with free variables or assumptions, open formulae occur routinely. We support this as a derived concept outside the kernel by commuting the future value with abstraction and application of local variables and assumptions, wrapping it into introduction / elimination rules of  $\wedge/\implies$ .

This auxiliary introduction of  $\beta$ -redexes into proof terms needs to be accounted as overhead for proof parallelization, but it is reasonably small for systems without explicit proof objects, like HOL and Isabelle. A version for Coq would require further attention to avoid losing efficiency here.

Some further layers of the prover require adjustment for parallel proofs, in particular the module for goal-directed proof and the Isar proof interpreter. Gladly this has relatively little impact for genuine user code, as can be seen for the Isabelle/ML function  $Goal.prove: Proof.context \rightarrow term \rightarrow tactic \rightarrow thm$ .

Here the tactic provides operational details to turn the given term into a theorem. We can provide the variant  $Goal.prove\_future$  with exactly the same signature: actual fork/join of future values is hidden in the implementation. There is a slight semantic difference, though, because future proofs assume that information about failed tactics can be deferred to the universal join in the end. Sometimes immediate feedback is more appropriate, cf. the monotonicity proof of **inductive** definitions which counts as a well-formed check of the specification. Any other derivations inside **inductive** can be deferred, though.

## 4 Performance and Scalability

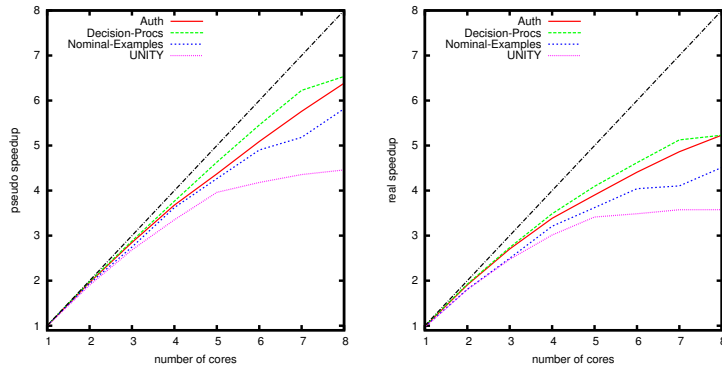
Above we have got some impression about various system layers involved in parallelization. Significant losses at any stage will accumulate and degrade the overall performance in this long chain. For example, a slowdown factor of 1.5 in the runtime system (due to reorganization of GC) will be hard to amend by parallelization with 2–4 cores, and getting real performance from 4–8 cores is even harder. Thus there is an important difference between experimental implementations of parallel languages, and ones that are fit for production use.

Since Poly/ML and Isabelle/ML are in the second category [12], we can face the even more difficult challenge to see how baseline performance scales towards more and more cores. To this end we pick a collection of Isabelle examples that have performed best, and look more closely when and how we get into scalability issues. Sequential runtime of these examples is in the range of 1.5–10 min:

<i>Auth</i>	278 s
<i>Decision-Procs</i>	453 s
<i>Nominal-Examples</i>	593 s
<i>UNITY</i>	95 s

The structure of these examples is quite diverse. *Auth* consists of a very broad graph of independent theories, each with many medium-sized toplevel proof scripts. *Decision-Procs* has fewer independent theory nodes, but many long proofs that are deeply structured in Isar. *Nominal-Examples* contains small unrelated examples and one big development consisting of a long sequence of long tactic scripts. *UNITY* is a mixture of small theories that lead up to the main theory, followed by various independent examples; most proofs are relatively short and weakly structured.

Measuring in the range of 1–8 cores we achieve the following speedups:



We see that the “pseudo speedup” on the left is not completely useless. It measures how much of the available CPU power is being used. Since our general CPU overhead wrt. sequential execution is relatively low (between 5% and 50% in worst case situations), there is a close correlation to the real speedup on the right. Real speedup of 5.2 for 8 cores is definitely in the top range of what is usually seen in the literature for large and complex applications, e.g. [10, 8, 11].

To understand what is missing for even better results, we analyze the degree of parallelism by monitoring the task queue population of our scheduler for future values (distinguishing proofs from other ready tasks), and the saturation of the worker threads (which correspond to real CPU cores). The results for *UNITY* (worst case) and *Decision-Procs* (best case) and are given in figure 2.

The task queue population evolves according to the  $q = 2$  strategy (§2.2): in the startup phase we process theories on the surface and accumulate many proof tasks; the final phase is dominated by proof processing, typically with a few long-running ones that determine CPU saturation at the tail end. Note that the additional non-proof tasks in the beginning stem from some peep-hole parallelization in parsing of Isabelle/Isar command and term language. This occasionally helps to fork more proofs more quickly.

The diagram shows how *UNITY* struggles to saturate the cores. In particular, the warm-up phase fails to produce sufficiently many proof tasks already with 4 cores: the relatively narrow theory structure produces too few proofs, or these are consumed too quickly. For 8 cores there are further drop-outs in the CPU saturation, but the elapsed runtime is only 27s anyway.

In contrast, *Decision-Procs* runs smoothly on 8 cores, due to its rich structure of theories and Isar proofs. A few remaining proof steps dominate the tail end, without keeping all cores busy. Despite the good CPU saturation for  $m = 8$ , further measurement for  $m = 16$  shows many drop-outs in the worker-thread saturation and a decline of the real speedup from 5.2 to 4.6.

Only *Auth* is able to gain further improvement when moving from  $m = 8$  with speedup 5.2 to  $m = 16$  with speedup 6.2, probably due to its unusually broad theory graph and substantial body of proofs. It will be interesting to see what fine-tuning in the sense of  $q = 3$  (cf. §2.2) and the next generation of hardware with 12–16 real cores can achieve for average Isabelle applications.

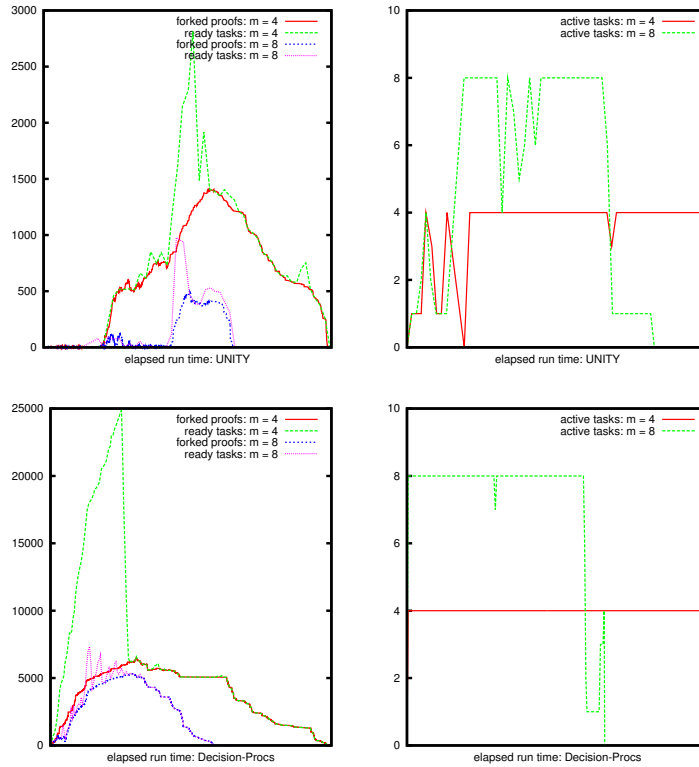


Fig. 2. Task queue population and worker thread utilization

## 5 Conclusion and Related Work

We have demonstrated that interactive theorem proving can make adequate use of the parallel computing power of current shared-memory multi-core hardware. Our main strategy for parallelization is based on the observation that formal theories consist of explicit statements with irrelevant proofs. Additional aspects of Isabelle/Isar sub-proofs are important for further scalability.

Implementations of parallel functional programming languages should in principle be commonplace, but we had to rebuild significant infrastructure for SML from scratch, based on parallel Poly/ML provided by David Matthews [19, 12]. Members of the HOL family that are also implemented in SML could re-use that immediately, but the genuine prover infrastructure would have to be reworked. In contrast, Coq faces more serious challenges since its home platform OCaml is highly-tuned for sequential execution, despite earlier work on parallel runtime systems for its predecessor Caml [3] that was later discontinued.

Provers implemented in LISP are a bit more fortunate, e.g. see the exploration of various possibilities for ACL2 [16], while ongoing work mainly targets speedup

of the immediate interaction process and co-operation of semi-automated proof tools. Haskell [8, 11] and Scala [15, 6] also provide dependable concepts for parallel functional programming that could be used for interactive theorem proving.

The challenge of explicit parallelism in application code has happened already in the late 1980-ies and early 1990-ies, when classic CISC machines became stagnant and “transputers” or workstation clusters were considered a viable perspective to gain more performance. Some parallel prover projects from that time include the *Distributed Larch Prover* [9], and the *MP refiner* [14] as parallel tactical engine for NuPrl, which was based on an experimental parallel version of SML/NJ [13]. This pioneering work had little impact on mainstream interactive provers later on, and the boost of performance of RISC machines and reformed CISC machines (with implicit instruction-level parallelism and large caches) has postponed the problem of everyday parallelization until 2005.

If cores really continue to multiply at an exponential rate, it means that our prover infrastructure needs keep up by more and more sophisticated parallelization strategies. We anticipate the following main lines of further improvements:

- internal parallelization of specific proof tools and decision procedures, also with parallel *proof search*, which can be modeled via exception propagation within groups of future tasks in Isabelle/ML;
- further sub-structural proof parallelization, exploiting virtues of the Isar proof language more thoroughly;
- integration of parallel checking into a document model for *asynchronous interaction*, as already sketched in our work on Isabelle/Scala/jEdit [20].

The latter idea aims at free-form editing and continuous proof checking of large formalizations. Thus interactive prover technology should eventually support truly pervasive parallelism of all aspects of theory processing, while the user is admitted to interact with this machinery asynchronously. This poses some challenges of instantaneous version control (within the editor history) and real-time conditions on prover feedback (for IDE-style presentation of results). Ultimately, we expect significant gain of efficiency from a shift of paradigms in prover interaction, not just from the underlying parallel hardware.

## References

- [1] H. Amjad. Shallow lazy proofs. In J. Hurd and T. F. Melham, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2005)*, volume 3603 of *LNCS*. Springer, 2005.
- [2] J. Armstrong, R. Viriding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.
- [3] D. Doligez and X. Leroy. A concurrent, generational garbage collector for a multithreaded implementation of ML. In *20th ACM Symposium on Principles of Programming Languages (POPL)*. ACM press, 1993.

- [4] M. Gordon, R. Milner, L. Morris, M. C. Newey, and C. P. Wadsworth. A metalanguage for interactive proof in LCF. In *Principles of programming languages (POPL)*, 1978.
- [5] M. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. LNCS 78. Springer, 1979.
- [6] P. Haller and M. Odersky. Scala actors: unifying thread-based and event-based programming. *Theoretical Computer Science*, 2008.
- [7] R. H. Halstead, Jr. Multilisp: A language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst.*, 7(4), 1985.
- [8] T. Harris, S. Marlow, and S. Peyton Jones. Haskell on a shared-memory multiprocessor. In *ACM SIGPLAN workshop on Haskell*. ACM Press, September 2005.
- [9] D. Kapur and M. T. Vandevoorde. DLP: A paradigm for parallel interactive theorem proving, 1996.
- [10] H.-W. Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16(3), 2003.
- [11] S. Marlow, T. Harris, R. P. James, and S. Peyton Jones. Parallel generational-copying garbage collection with a block-structured heap. In *International Symposium on Memory Management*, 2008.
- [12] D. Matthews and M. Wenzel. Efficient parallel programming in Poly/ML and Isabelle/ML. In *ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010)*, 2010.
- [13] J. G. Morrisett and A. Tolmach. Procs and locks: a portable multiprocessing platform for Standard ML of New Jersey. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP)*. ACM, 1993.
- [14] R. Moten. Exploiting parallelism in interactive theorem provers. In J. Grundy and M. C. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs 1998)*, volume 1479, 1998.
- [15] M. Odersky et al. An overview of the Scala programming language. Technical Report IC/2004/64, EPF Lausanne, 2004.
- [16] D. L. Rager. Adding parallelism capabilities in ACL2. In *ACL2 '06: Proceedings of the sixth international workshop on the ACL2 theorem prover and its applications*. ACM, 2006.
- [17] H. Sutter. The free lunch is over — a fundamental turn toward concurrency in software. *Dr. Dobbs's Journal*, 30(3), 2005.
- [18] M. Wenzel. Isabelle/Isar — a generic framework for human-readable proof documents. In R. Matuszewski and A. Zalewska, editors, *From Insight to Proof — Festschrift in Honour of Andrzej Trybulec*, volume 10(23) of *Studies in Logic, Grammar, and Rhetoric*. University of Białystok, 2007.
- [19] M. Wenzel. Parallel proof checking in Isabelle/Isar. In G. Dos Reis and L. Théry, editors, *ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009)*. ACM Digital Library, 2009.
- [20] M. Wenzel. Asynchronous proof processing with Isabelle/Scala and Isabelle/jEdit. In C. Sacerdoti Coen and D. Aspinall, editors, *User Interfaces for Theorem Provers (UITP 2010)*, ENTCS, July 2010. FLOC 2010 Satellite Workshop.
- [21] M. Wenzel. *The Isabelle/Isar Implementation Manual*, 2011.
- [22] M. Wenzel and B. Wolff. Building formal method tools in the Isabelle/Isar framework. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics (TPHOLs 2007)*, volume 4732 of *LNCS*. Springer, 2007.
- [23] F. Wiedijk, editor. *The Seventeen Provers of the World*, volume 3600 of *LNAI*. Springer, 2006.