Shared-Memory Multiprocessing for Interactive Theorem Proving

Makarius Wenzel *

Univ. Paris-Sud, Laboratoire LRI, UMR8623, Orsay, F-91405, France
CNRS, Orsay, F-91405, France

Abstract. We address the multicore problem for interactive theorem proving, notably for Isabelle. The stagnation of CPU clock frequency since 2005 means that hardware manufactures multiply cores to keep up with "Moore's Law", but this imposes the burden of explicit parallelism to application developers. To cope with this trend, Isabelle has started to support parallel theory and proof processing in 2007, and continuously improved the use of multicore hardware in recent years. This is of practical relevance to theory and proof development, since their size and complexity is roughly correlated with the real time required for re-checking. Scaling up the prover on parallel hardware will facilitate maintenance of larger theory libraries, for example.
Our approach to parallel processing in Isabelle is mostly implicit, without user intervention. The system is able to exploit the inherent problem-structure of LCF-style proof checking, although it requires substantial reforms of the prover architecture and its implementation. Thus the user gains significant speedup factors on typical commodity hardware with 2–32 cores; saturation of 8 cores is already routine in many applications. The present paper provides an overview of the current state of shared-memory multiprocessing in Isabelle2013, which also benefits from recent improvements of parallel memory management in Poly/ML (by David Matthews). We discuss common requirements, problems, and solutions. Concrete performance figures are analyzed for some applications from the Isabelle distribution and the Archive of Formal Proofs (AFP).

1 Introduction

1.1 The Multicore Problem

Software developers have become accustomed to *Moore's Law* of computing, which states that chip density and integrated functionality doubles every two years. This is essentially a social contract of hardware manufactures with its customers, the producers of computer systems and application software. In the past, it was correlated with an increase in clock frequency, so existing programs would become exponentially faster over time, or the complexity of programs could be increased without the user noticing such "software bloat".

* Current research supported by Project Paral-ITP (ANR-11-INSE-001).

The rules have changed substantially around 2005, when clock frequency has reached a plateau at 3 GHz — due to excessive power dissipation and overheating at higher rates. Thus the continued exponential growth leads to a multiplication of explicitly visible CPU cores, presently in the range of 2–32.

Multiplication of cores naturally poses challenges to application software development. Sequential code that fails to adapt to this evolutionary pressure suffers from exponential decline of relative performance. For example, a single-core program on 16 cores is confined to 6.25% of the nominal CPU power. Even if the number of cores becomes stagnant, which might well happen especially in the consumer market, we are left with the problem of a large gap in potential application performance, and the era of sequentialism is not coming back.

Multiprocessing does not provide "spare CPU cycles" for free: extra effort is required to use the available CPU power in applications. Performance matters for interactive theorem proving, since the real time spent for re-checking is correlated with the total size of formal developments. Big Isabelle applications (e.g. from the Archive of Formal Proofs) typically grow until the time for full re-checking approaches 10 min to 1 h. It is up to the prover implementation to stretch the amount of formal content that can be processed in that time-span.

An important characteristic of the hardware class with 2–32 cores is that the convenient programming model of *shared memory* can still be supported, with reasonable memory bandwidth for transfers between CPU modules. This requires hardware manufacturers to provide an increasingly complex memory hierarchy of caches and quick paths for physically distributed memory, but it is one area where the exponential increase of hardware capabilities still happens (apart from graphics performance). There is some variance in the different product lines of Intel vs. AMD: in 2013 high-end CPUs by Intel emphasize the performance of shared memory access, while AMD maximizes the number of cores per chip.

For the concrete measurements in this paper, 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 Mountain Lion in genuine 64 bit mode. This represents a typical (slightly dated) workstation. The performance of current high-end laptops (e.g. based on Intel Core i7) is only a factor of 2 lower than that — there is usually just one CPU module on mobile systems.

1.2 LCF-style Provers as Multi-threaded Applications

Due to various characteristics, interactive theorem provers like HOL [19, §1], Coq [19, §4], Isabelle [19, §6], or ACL2 [19, §8] fit quite well into the 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 λ-terms for syntax or proof terms).

In the multicore era, immutability is one of the inherent advantages of pure functional programming, and even Java programmers have noticed that (cf. the attention that functional-object-oriented Scala and the LISP/Haskell dialect Clojure have gained in the JVM world). Shared memory allows to pass pointers to immutable data without requiring copying, and without the 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. This is exploited in the parallel garbage collector of Glasgow Haskell [8] and Poly/ML. Thus pure functional programs can afford thread-based parallelism, without the hazards known from C or FORTRAN. Nonetheless, threads and synchronization primitives are difficult to use directly in application code, so higher principles of parallel functional programming are required.

LCF-style abstraction of formally certified entities. In the LCF architecture [5] that is observed by the HOL family and Isabelle, certified entities like theorems, well-formed terms, and theory certificates are represented as entities of abstract datatypes. The corresponding proof constructions only exist as Platonistic ideas, without 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 requires externalization of formal entities. Depending how thoroughly proof checking is treated at the kernel level, this may demand full proof terms to be communicated, say over a network of CPUs.

Continuous interaction with a large prover process. Our prover interaction model is centered around a single process with a large background context, where the user produces small additions incrementally. This scenario can be efficiently represented by a single multi-threaded process.

Using separate prover processes instead, say via Unix-fork with the usual "copy-on-write" implementation of virtual memory, is faced with some problems. First, the initially shared physical memory map diverges after some run-time of the ML system, notably due to garbage collection that moves equal content in different ways and thus produces separate copies. Second, the results of a fork need to be communicated back by explicit inter-process communication, using some externalized form of proof objects. On non-Unix systems (Windows), startup time of a fresh prover process might be even considered too high in immediate user interaction.

In contrast, the advanced parallel memory management of multi-threaded Poly/ML retains the original structure of data on a shared heap. Recent Poly/ML 5.5 explicitly recovers structural sharing of equivalent data in a long-running ML process, as a special phase in its parallel garbage collection. This allows to scale to more threads working on less memory: 32 bit mode with small 2–3 GB address space has become interesting again for big Isabelle applications, due to reduced memory bandwidth requirements.

These introductory observations should make sufficiently clear that interactive theorem proving and shared-memory multiprocessing are worth investigating and turning into practice. 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 could be transferred to other interactive provers.

2 Strategies for Parallel Proof Checking

Subsequently, the running example is the medium-sized entry *Slicing* from AFP (http://afp.sf.net/entries/Slicing.shtml). Its sequential runtime is 12 min.

2.1 Peep-hole Parallelism

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

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

This solves a goal state of 1225 subgoals by simplification in 77 s elapsed time. Since all subgoals are independent, without any schematic variables whose instantiation could influence each other, it is an "embarrassingly parallel problem". The obvious idea is to simplify these subgoals separately and recombine the results by back-chaining with the original goal state. In Isabelle2013, the proof method *simp_all* is smart enough to detect this situation and to operate in parallel by default. It uses the general-purpose tactical `PARALLEL_GOALS`, based on `Par_List.map` in Isabelle/ML. 1225 simplification tasks are forked, and all results joined before proceeding. Timings for this experiment are given in figure 1.

worker threads m	elapsed time $\varepsilon(m)$	CPU time $\zeta(m)$	pseudo speedup $\zeta(m)$ / $\varepsilon(m)$	real speedup $\varepsilon(1)$ / $\varepsilon(m)$
1	77.0 s	77.3 s	1.0	1.0
2	38.5 s	76.5 s	2.0	2.0
4	19.7 s	76.8 s	3.9	3.9
6	13.8 s	79.5 s	5.8	5.6
8	10.7 s	80.0 s	7.5	7.2
12	9.1 s	99.1 s	11	8.5
16	8.1 s	113 s	14	9.5

Fig. 1. Simplification of 1225 independent subgoals, with hyperthreading for $m > 8$.

Empirical results of parallel performance need to be treated carefully, looking closely what is measured and how. The figures of elapsed time vs. CPU times are based on standard facilities of the operating system, which we take for granted. The column "pseudo speedup" gives some impression how much nominal CPU cycles are spent, but generally does not tell what the user gains (apart from

extra heat production by the computer). The "real speedup" $\varepsilon(1)$ / $\varepsilon(m)$ represents the success of parallelization more faithfully, but it is usually lower and less exciting in the presentation, and $\varepsilon(1)$ is unknown in practice without the sequential run for comparison. Isabelle batch mode displays the ratio $\zeta(m)$ / $\varepsilon(m)$ as speedup factor by default, and it happens to approximate the real speedup above reasonably well, before the multicore system is pushed towards its limits.

What have we gained so far? Reducing a single tactic application from roughly 80 s to 8 s gives an impressive speedup factor 10, an isolated boost of performance that might be useful for the user working at that spot. Looking at the bigger picture, though, the overall runtime of AFP/Slicing merely shortens from 720 s to 648 s (factor 1.1), as there are no further "embarrassingly parallel problems". This means the total performance improvement with 16 worker threads is not 1000%, but 10%. In other applications it might be as low as 1%.

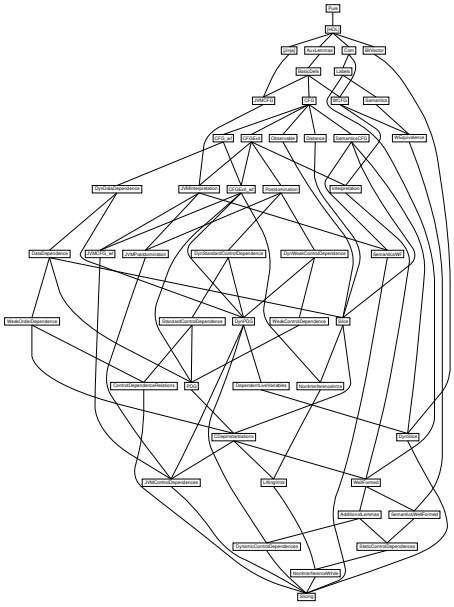
This effect is typical for "peephole parallelization", it applies to most problem domains when the aspect of parallelism is considered naively. *Amdahl's Law* (from 1967) estimates the sub-linear speedup as $1 \;/\; (s + p/m)$, where s is the part of the program running sequentially, and p the part running in parallel (normalized such that $s + p = 1$). For $m \longrightarrow \infty$ this converges to $1 \;/\; s$. In other words, the overall success of parallelization depends on the remaining fraction of inherently sequential code. The prediction would become more pessimistic by including losses due to organization of parallel computation, so for large number of cores the speedup eventually becomes smaller than 1, and ultimately tends towards 0.

2.2 Pervasive Theory and Proof Parallelization

The main conclusion of the previous experiment is that parallelism needs be *pervasive* to gain significant speedup, i.e. the remaining sequential part of the application runtime needs to approach 0. In 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.

Granularity q = 0: parallel theories.
Typical formalizations consist of an acyclic graph of theory nodes, often with a reasonable degree of independent paths, e.g. see figure 2 (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 resembles `make -j` on the Unix command-line, but we run multiple threads within the same ML process, using our own scheduler for DAG-structured evaluation in Isabelle/ML.

Granularity q = 1: parallel theories and toplevel proofs.
As sketched in figure 2 (right), each theory node consists of a sequence of definition–statement–proof. Results are *specified* beforehand as propositions in the text, and later *justified* by the proofs, which are *irrelevant* in practice. Likewise, some definitional forms require proofs internally (e.g. **inductive**).



inductive P ...

theorem A ⟨*proof*⟩
theorem B ⟨*proof*⟩
theorem C ⟨*proof*⟩

have $X \wedge Y$
proof
  show X **by** *simp*
  show Y **by** *blast*
qed

theorem A_1 **and** ... **and** A_n
  by *simp_all*

Fig. 2. Typical theory dependencies and content structure.

Even though proofs are hard to produce and take long to check, they are not required to process the outermost sequence of specifications. Proofs can be forked immediately and are only joined in the very end, when the whole graph of loaded theories is consolidated. Proofs may refer to previous theorems, but not to their proofs.[1]

So the totality of proofs emerging from a given theory graph poses again some "embarrassingly parallel problem" as in §2.1, but with slightly different characteristics: proof problems emerge dynamically during the ongoing theory processing, and need to be joined only in the very end. Instead of static skeletons like `Par_List.map`, the appropriate programming model is that of dynamic *fork* / *join* of eventual results (cf. §3.1).

Granularity q = 2: parallel theories, toplevel proofs, and end-proofs in Isar.
Incidentally, the structured proof language Isabelle/Isar [16] provides high degree of compositionality, and thus extra potential for parallel checking. In principle, every Isar sub-proof could be treated recursively like $q = 1$, but for simplicity we only do this for Isar end-proofs "**by** *method*". This is sufficient for structured 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.).

Concrete results for these parallelization strategies are given in figure 3. In each column for q, the speedup curve for increasing m flattens according to

[1] Despite the different foundational approach in the Dependent Type Theory of Coq, proof irrelevance still holds in practice, since most proofs are "opaque".

Amdahl's Law. This is inevitable, but it matters how far the relative decline can be postponed. The combination of parallelization strategies for $q = 2$ achieves fairly good speedup of 6.2 on 8 cores, more than 75% of the nominal CPU power.

worker threads m	real speedup $q = 0$	real speedup $q = 1$	real speedup $q = 2$
2	1.6	1.8	2.0
4	2.0	3.0	3.3
6	2.1	3.5	4.1
8	2.2	3.7	6.2

Fig. 3. Real speedup of AFP/Slicing depending on granularity q.

The results of this single experiment can be extrapolated — it has been chosen to represent typical Isabelle applications seen today. Thus we conclude the following rules of thumb for multicore scalability:

- Parallel theory loading alone scales to **2 cores**,
- with additional toplevel proof parallelization it scales to **4 cores**,
- with additional sub-structural proof parallelization it scales to **8 cores**.

Another parameter that is not measured here is the level of sophistication of the parallel prover implementation. In 2011 AFP/Slicing did not scale beyond 4 cores, and in 2009 only few Isabelle applications managed to go significantly beyond 2 cores — see [15, §5] for the best that could be achieved on 4-core Intel Xeon hardware in that time.

3 Parallel Prover Architecture

Despite good side-conditions for multi-threaded proof checking (§1.2), substantial reforms of the prover architecture (and its implementation) are required to make it actually work and perform well. This affects a broad spectrum of core prover aspects: parallel functional programming, parallel inference kernel, explicit organization of theory and proof structure.

There is further impact on outer system integration layers. For example, Isabelle2013 provides an advanced build system (implemented in Isabelle/Scala) to manage re-checking of large theory libraries efficiently, by managing a tree of multi-threaded processes that run in parallel. Thus by exploiting the outer hierarchy of "sessions" and the inner structure of theories and proofs, full re-checking of AFP has been reduced from several hours to 30 min on 8 cores.

Further integration of parallel checking and asynchronous interaction happens in the Prover IDE [17]. In Isabelle2013 it exploits more fine-grained proof parallelism during regular interaction. Combining erratic edits by the user and continuous parallel checking by the prover poses further challenges that are beyond the scope of the present paper (some aspects are discussed in [18]).

Subsequently, we provide an overview of Isabelle2013 prover architecture, putting it into perspective of earlier work and pointing out recent refinements.

3.1 Parallel ML

LCF-style theorem proving has been intertwined with functional programming in ML since the inception of both in LCF [4]. Originally implemented as an interpreted language within LISP, ML has become a standalone language in the 1990-ies. Some of its implementations have managed to catch up with multithreading already, such as Poly/ML [9] (with its own parallel runtime system) and F# (which re-uses the common language runtime of the .NET platform). Other interesting functional languages with good support for parallel programming are Haskell [8] and Scala, and of course LISP where important ideas of task-parallel evaluation was first explored in the 1980-ies [6].

The most notable exception is OCaml, which is still subject to early decisions by its main architects of *not* supporting parallel threads in ML, despite [3].

The host language for particular provers is inherited from distant past and somehow accidental, but it impacts chances of survival in the multicore era. Isabelle has always supported more than one implementation of Standard ML, especially Poly/ML and SML/NJ. Starting in 2006, David Matthews made substantial renovations for Poly/ML to support native multithreading. The Poly/ML 5.5 release from September 2012 is notable for its support of parallel garbage collection and compaction of large heaps. There is now a considerable performance gap towards SML/NJ: in 2013 the factor is of the order 10^2 for medium-sized Isabelle applications, and big ones are already infeasible.

In a system like Poly/ML, the raw power of shared-memory multicore hardware is made available as *threads–and–locks*. Poly/ML offers an ML view on POSIX threads, with its mutexes and condition variables for synchronization and signaling [9, §2]. This first approximation to parallel computation is then augmented by a concept for task parallel programming which organizes evaluation of *future values* in Isabelle/ML [9, §3]. Threads do not scale beyond 10^1–10^2, but a limited number of worker threads can operate efficiently on a task queue of 10^5–10^6 pending evaluations. The idea of data-oriented parallelism dates back to Multiplisp [6] at the least. It has been re-implemented over decades in many variations, and is routinely available in Isabelle/ML, F#, Haskell, Scala, LISP.

The Isabelle/ML implementation of futures is careful to transfer the semantics of Standard ML adequately into the parallel environment, with strict functional evaluation, synchronous program exceptions, and asynchronous interrupts. The main programming interface is as follows:

type α *future*
val *Future.fork*: $(unit \to \alpha) \to \alpha$ *future*
val *Future.join*: α *future* $\to \alpha$
val *Future.cancel*: α *future* $\to$ *unit*

Type α *future* represents the eventual result of a given expression, which is associated with an evaluation task of the future scheduler in the background. The task queue supports both priorities and dependencies, and is implemented as explicit graph structure. Joining with an unfinished future synchronizes with the evaluation process, where the full complexity of inter-thread communication happens. In contrast, there is no special overhead to access finished futures later.

The key property is that *Future.join* (*Future.fork* (**fn** () $\Rightarrow$ *expr*)) produces the same result (or exception) as *expr* outright, but the fork can be separated from the join for parallel evaluation. The overhead for fork + join is about $10^{-5}\,s$. On Intel hardware similar to our's, Rager [13] reports 50 μs for his LISP system, and we measure exactly the same for Isabelle/ML. This overhead roughly determines the granularity of tasks that are feasible to fork. For example, 20000 tasks that run a few microseconds each will waste 1 s, but this not a problem if the application manages to produce thousands of tasks in the range of milliseconds.

Futures can be used to implement higher-level combinators like parallel `map` (or the renowned `reduce`). Isabelle/ML already provides such derived combinators, but proof parallelization uses *Future.fork* and *Future.join* directly, because proofs emerge dynamically during the exploration of yet unknown theory content. Since forks are under program control, we can exploit potential parallelism while exploring the proof, and easily restrain substructural parallelization tasks.

3.2 Theory Context and Proof Promises

Logical derivations work in a context, whose precise structure depends on the formulation of the underlying logic. In the HOL family and in Isabelle, there is a *global theory context* that we call Θ, and a *local proof context* that we call Γ.

The theory Θ contains declarations and specifications of type constructors, term constants, and axioms (definitions), which are polymorphic in the sense that their type schemes may get instantiated arbitrarily during proof.

The proof context Γ contains local hypotheses (premises) to support $\Longrightarrow$ introduction in Natural Deduction. Locally fixed parameters for $\bigwedge$ quantifier introduction are implicit (in contrast to dependent type theory in Coq).

The inference kernel produces sequents Θ, $\Gamma \vdash \varphi$, but we have $\Gamma = \emptyset$ for global results and the background theory Θ is managed implicitly. Thus end-users may think just of theorems $\vdash \varphi$ that establish a certain proposition φ.

Nonetheless, the global context Θ turns out as essential for management of forked proofs in parallel Isabelle. Lets say that at stage Θ_1 of the ongoing theory development, theorem φ is claimed and its proof forked for independent checking, while the theory is continued monotonically towards Θ_2, adding more definitions and theorems. When the forked proof is eventually joined, it needs to establish $\Theta_1 \vdash \varphi$ in the original theory context for proper foundation of logical results.

This means the inference kernel needs to work explicitly with theory contexts, with some operations to extend, merge, compare theories according to $\Theta_1 \subseteq \Theta_2$, and transfer of theorems from the smaller to the bigger theory.

Incidently, Paulson [11] had already introduced a notion of theory context for theorems in Isabelle89. This was motivated by the *logical framework* approach of that time, to allow the user to work in different background contexts. The concept has been refined many times, notably for efficient checking of $\Theta_1 \subseteq \Theta_2$ via symbolic theory certificates that represent the stages of extend and merge operations, without inspecting the theory content directly.

Another important aspect is extra-logical *theory data*: concrete syntax, hints for proof tools etc. are managed in a value-oriented manner as part of Θ (and Γ) in the implementation. Thus tools may run in parallel and refer safely to their private data within the context, without worrying about mutable state.

This is in contrast to original LCF and members of the HOL family, who accumulate theory content as implicit global state of the ML process. The state grows monotonically as the user adds new definitions, without returning to earlier states (undo) and without isolation of independent instances of proof tools. Coq is slightly more flexible by multiplexing different contexts, with operations *freeze* and *unfreeze* for tool hints that are associated with them, although this concept is still restricted to a single active view of the state.

These observations reveal accidental side-conditions in the long history of LCF-style provers. In the past, the interaction model was that of a single-threaded TTY loop where individual commands were applied one after another, so simplistic treatment of the context could be afforded. But this should not prevent Coq and HOL systems to become more stateless and timeless eventually.

Taking sequents $\Theta, \Gamma \vdash \varphi$ with explicit context values for granted in Isabelle, we can proceed in the next stage to support a notion of *proof promises* natively in the inference kernel. The new rule *promise* produces a hole (with specified result) in the reasoning, which can be amended later by another rule *fulfill*. Holes are managed formally by the context Π that maps identifiers of proof promises to actual derivations. The original version from 2008 of this slightly extended Natural Deduction system for Isabelle/Pure is given in [15, §3]; according to Isabelle2013 the main rules are as follows:

$$\frac{\mathrm{FV}\, A = \emptyset \quad \mathrm{TV}\, A = \{?\overline{\alpha}\}}{\Theta, \{a : A\}, \emptyset \vdash a[?\overline{\alpha}] : A[?\overline{\alpha}]}\ (\mathit{promise})$$

$$\frac{\Theta, \Pi, \Gamma \vdash p : B \quad \Theta_0, \emptyset, \emptyset \vdash q : A \quad \Theta_0 \subseteq \Theta}{\Theta, \Pi - \{a : A\}, \Gamma \vdash p[a := q] : B}\ (\mathit{fulfill})$$

The underlying formulation of Isabelle/Pure with proof terms goes back to [2]; it emphasizes the role of proof promises as polymorphic proof constants, which may be substituted by closed proof terms later. In reality, proof terms are merely a second option of the Isabelle kernel. By default it only maintains a *proof body* as a digest of proof promises, oracles, and external theorem references.

A notable refinement of *fulfill* compared to [15, §3] is that the replacement proof $q : A$ is required to be fully closed ($\Pi = \emptyset$). This avoids complications of holes depending on other holes, and speculative well-founded ordering of the same, which would be hard to implement in practice. The restricted form means that future proofs need to be joined in a bottom-up manner before passing through the inference kernel. The well-founded order is given by the physical process of Isabelle/ML consolidating values. In best LCF tradition, this might lead to non-termination, but cannot produce unfounded results.

Management of free type variables is explained further in [15, §3]. It is possible to quantify term variables and thus demand $\mathrm{FV}\ A = \emptyset$ w.l.o.g., but type

variables need to be tracked separately to retain the schematic polymorphism of the logic. Otherwise it would be impossible to fork a proof and instantiate types of the result. Our context Π acts like Θ in this respect. This is in contrast to shallow proof holes of [1], which are monomorphic due to the use of Γ in HOL.

The ML interface of the inference kernel presents *promise* and *fulfill* together as a single operation *Thm.future*: *thm future* $\rightarrow$ *term* $\rightarrow$ *thm*. It makes a theorem based on a proof promise that will be fulfilled by the eventual result of the given future. The side-conditions are checked on finished future derivations. Thus the implicit policy of future evaluation is re-used, but the kernel stays in control of checking the outcome as plain ML value; it does not even care about parallelism.

ML values of type `thm` are theorems on the surface only, as they may depend on unfinished futures. The kernel operation *Thm.join_proofs*: *thm list* $\rightarrow$ *unit* consolidates results by recursive joining of the graph of pending proof promises, and produces a digest of parallel error messages about failed proofs.

3.3 Goals with Forked Proofs

The notions of proof promises and future theorems of the inference kernel mainly serve foundational purposes in the spirit of the LCF architecture. The main programming interface works via goals with forked proofs. There is some additional infrastructure for accounting and reporting of structural errors within forked proofs. As already observed in [15, §4], the Isabelle/ML operation for goal-directed proof *Goal.prove*: *Proof.context* $\rightarrow$ *term* $\rightarrow$ *tactic* $\rightarrow$ *thm* can be turned into an alternative version *Goal.prove_future* of the same signature. Its internal use of future theorems is hidden, thanks to the full specification of the intended result as proposition. This is a key advantage of backward-proof.

There are delicate differences in the semantics of proof failure, though, if errors in forked proofs are postponed until the final join over all theories. This is important for derived definitional packages like **inductive** in Isabelle/HOL, where failure of its internal monotonicity proof means that the user specification is malformed. Any further derivations inside **inductive** are irrelevant to the user: they always work under the assumption that the package is implemented properly. This critical treatment of forked proofs was still relatively crude in 2008 [15], resulting in more conservative use of sequential proofs in some situations.

To avoid such conflicts of parallelism and reliability of the prover, the high-level infrastructure for goal-directed parallel proof has been reworked significantly for Isabelle2013. The main aspects are summarized as follows:

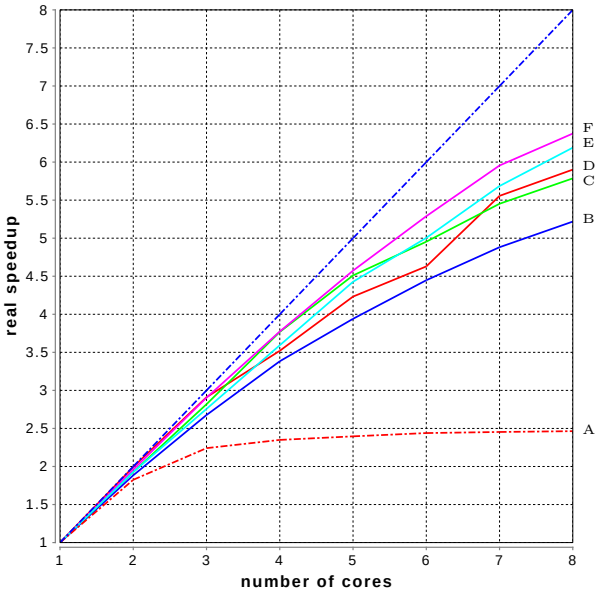
- λ-lifting wrt. to the proof context Γ according to [15, §4.1], to allow goal statements work with premises and parameters, despite the restrictions of *Thm.future_thm* due to the *promise* and *fulfill* rules (§3.2).
- Global accounting of forked proofs within the running process, to avoid unnecessary forks when the system is flooded with future tasks already, according to the bound of $m *$ *parallel_proofs_threshold* (default 100).
- Systematic tracking of errors stemming from forked proofs according to the originating command transaction.

Consequently, derived elements like **inductive** and **datatype** may now fork their internal proofs more aggressively, even relevant ones. This is especially important for increased parallelism in the asynchronous document model of the Prover IDE. An explicit notion of *stable command* in its document model indicates the status of all goal forks, without requiring a global join. The system will reset failed command transactions whose forked proofs were failing or interrupted, and thus retry evaluation in the next editing phase.

4 Performance and Scalability

Asymptotically, the multicore problem cannot be solved, but we do our best to exploit the capabilities of our hardware. Subsequently we review further results of measuring the parallel performance, to see trends beyond 8/16 cores. Current Poly/ML 5.5 and Isabelle2013 allow monitoring of CPU and memory usage, status of parallel garbage collection, future tasks and worker threads.

For users the main result is the real speedup $\varepsilon(1)\ /\ \varepsilon(m)$, which is presented in figure 4 for various Isabelle sessions.



		$\varepsilon(1)$	$\varepsilon(8)$
A	*HOL*	210 s	85 s
B	*HOL-UNITY*	69 s	13 s
C	*HOL-Nominal_Examples*	526 s	91 s
D	*Slicing*	720 s	122 s
E	*HOL-Decision_Procs*	415 s	67 s
F	*HOL-Hoare_Parallel*	218 s	34 s

Fig. 4. Sequential runtime and real speedup of some Isabelle sessions

Session HOL is special here in compiling big ML modules for tools like Sledgehammer, and comparatively few regular theory and proof developments. Factor 2.5 for 8 cores might look disappointing, but it is already an improvement over 1.7 in [15, §5], where the base-line performance was much lower as well.

The other sessions are more conventional, with good speedup in the range of 5.2–6.4 for 8 cores. To scale further, potential losses in the implementation

of the parallel ML infrastructure are only of minor concern: they can be ironed out eventually. The main challenge is proper partitioning of parallel tasks according to the structure of the application. The histograms in figure 5 illustrate distribution of the runtime of tasks for $m = 8$ and $q = 2$ (cf. §2.2).

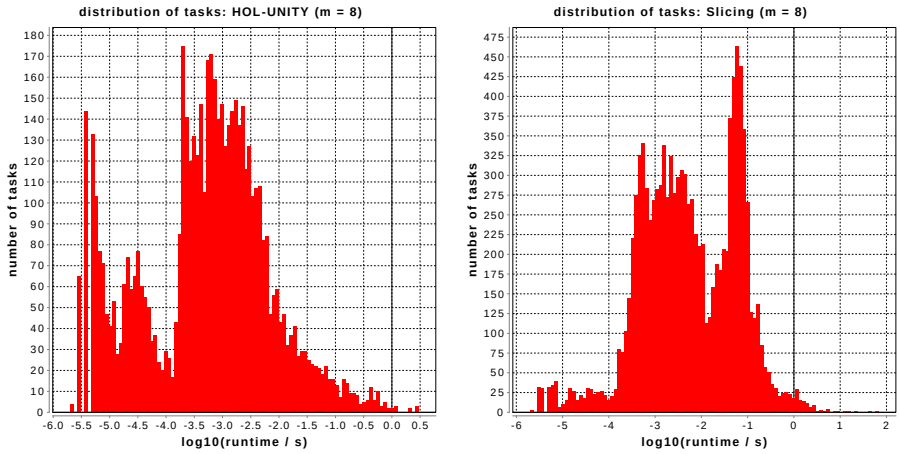


Fig. 5. Distribution of task runtimes ($m = 8$)

The examples HOL-UNITY vs. AFP/Slicing are very different in their absolute runtime and overall structure of theories and proofs. Both scale to 8 cores, which can be explained by a good amount of tasks in the millisecond range. AFP/Slicing tends to more longer-running proofs (many slow automated steps), with a few monolithic tasks in the range 10^1–10^2 s.

In figure 6 we see more precisely how this portfolio of future tasks populates the Isabelle/ML task queue and corresponding worker threads, over the elapsed runtime of each session. This changes significantly for different values of m.

The fluctuation of thousands of ready tasks is mainly due to forked proofs. These easily saturate 4 worker threads during most of the elapsed runtime, but for 8 there are increasing drop-outs with inactive workers: HOL-UNITY has a slow startup-ramp, until sufficiently many proof tasks are forked; AFP/Slicing has a slow tail-end with a few long-running tasks. The final theory of AFP/Slicing consists of one huge proof, with many automated steps; this is where many of its 10^1–10^2 s tasks emerge. The sudden peak near the start of AFP/Slicing is due parallel checking of 1225 subgoals, as discussed in §2.1.

5 Conclusion

We have demonstrated that proof checking in the LCF tradition can exploit the computing power of shared-memory multicore hardware adequately. 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.

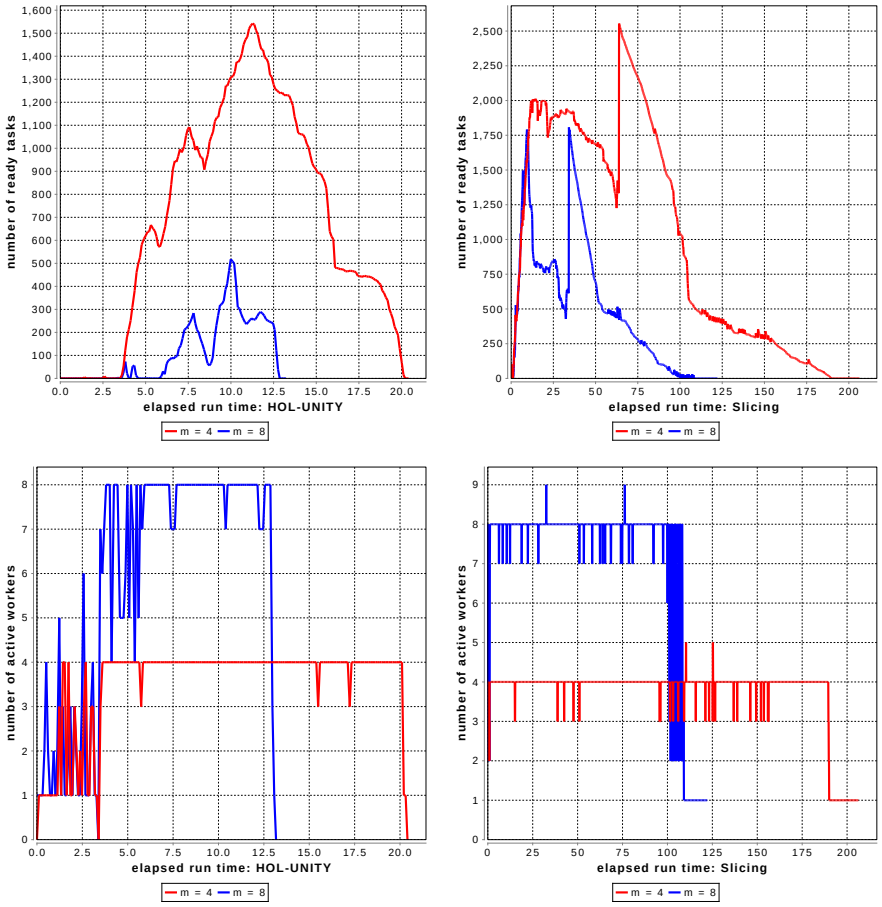


Fig. 6. Task queue population and worker thread utilization ($m = 4$ and $m = 8$)

Implementations of parallel ML should in principle be commonplace, but we had to rebuild significant infrastructure for SML from scratch (starting in 2006/2007), based on parallel Poly/ML provided by David Matthews. Versions of HOL that are also implemented in SML could re-use that, but some prover-specific infrastructure needs to be reworked. Coq faces more serious challenges since its home platform OCaml is optimized for sequential execution, although its predecessor Caml once had a multithreaded runtime system [3].

To relate performance figures of earlier versions of parallel Isabelle, note that [15, §5] also uses the top-end Mac Pro at that time (4-core Intel Xeon), but [9, §5] is different in using a fat-node of a computing-cluster (32 AMD Opteron CPUs and 64 GB main memory); the measurements of [9, §5.3] stretch the available resources, to explore the limits of ML memory management and application task structure [9, §5.3]. The present results require much less memory, and work even within the restricted address space of the 32-bit version of Poly/ML 5.5.

The challenge of explicit parallelism in application code has happened before in the late 1980-ies and early 1990-ies, when classic CISC machines became stagnant and "transputers" or workstation clusters were considered a viable alternative to gain more performance. Some parallel prover projects from that time include the *Distributed Larch Prover* [7], or the *MP refiner* [10] as parallel tactical engine for NuPrl, using an extinct parallel version of SML/NJ. This pioneering work had little impact on mainstream interactive provers later on, and the boost of performance of RISC machines and reformed CISC machines has postponed the problem of mainstream parallelism until 2005.

The only other major proof assistant that answers the multicore challenge is ACL2: Rager [12, 13, 14] provides parallel execution within the LISP system, and reworks the main stages of the interactive proof development process ("the ACL2 waterfall") to support parallelism in many practical situations. Performance is evaluated into the range of 32 cores on latest Intel hardware. ACL2 6.0 from December 2012 includes the parallel variant ACL2(p) already.

ACL2(p) emphasizes parallel enhancement of interactive proof discovery and case-splitting. This roughly corresponds to our sub-structural Isar proof parallelization, as far as it is already supported in Isabelle2013, but the side-conditions of the proof languages are quite different. Isabelle/Isar emphasizes fast rechecking of structured proof texts, while ACL2(p) emphasizes the search involved in its "waterfall" of interactive proof exploration.

As cores continue to multiply at an exponential rate, our prover infrastructure needs to catch up by more sophisticated parallelization strategies: the multicore problem poses a new challenge for every power of 2 in the CPU multiplication phenomenon. The advanced monitoring facilities of Isabelle2013 will help to isolate bottle-necks in the granularity of future tasks.

We anticipate further sub-structural proof parallelization, exploiting virtues of the Isar proof language more thoroughly: recursive proof forking to accommodate nested Isar proofs that spend substantial time in their outline structure. Another possibility is to introduce more explicit parallelism in specific proof tools, including parallel proof search. Isabelle/ML already provides combinators like `PARALLEL_CHOICE` for tactics, or `Par_List.exists` for generic ML functions.

As the Isabelle Prover IDE manages to support more and more parallelism in its asynchronous editing process, we expect significant shifts of paradigms how large proofs are developed, beyond the raw speedup from the underlying parallel hardware. This requires users to get acquainted with a timeless and stateless model of document-oriented proof development, and to give up manual control in "driving" the prover in single sequential steps. An advanced proof assistant acts like system software in this respect, and does the parallel scheduling implicitly and automatically without user intervention.

References

[1] Amjad, H.: Shallow lazy proofs. In Hurd, J., Melham, T.F., eds.: Theorem Proving in Higher Order Logics (TPHOLs 2005). Volume 3603 of LNCS., Springer (2005)

[2] Berghofer, S.: Program extraction in simply-typed Higher Order Logic. In Geuvers, H., Wiedijk, F., eds.: Types for Proofs and Programs (TYPES 2002). LNCS 2646, Springer (2003)
[3] Doligez, D., Leroy, X.: 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] Gordon, M., Milner, R., Morris, L., Newey, M.C., Wadsworth, C.P.: A metalanguage for interactive proof in LCF. In: Principles of programming languages (POPL). (1978)
[5] Gordon, M.J.C., Milner, R., Wadsworth, C.P.: Edinburgh LCF: A Mechanized Logic of Computation. Volume 78 of LNCS. Springer (1979)
[6] Halstead, R.H.: Multilisp: A language for concurrent symbolic computation. ACM Trans. Program. Lang. Syst. **7**(4) (1985)
[7] Kapur, D., Vandevoorde, M.T.: DLP: A paradigm for parallel interactive theorem proving (1996)
[8] Marlow, S., Peyton Jones, S.L., Singh, S.: Runtime support for multicore Haskell. In Hutton, G., Tolmach, A.P., eds.: 14th ACM SIGPLAN international conference on Functional programming (ICFP 2009), ACM (2009)
[9] Matthews, D., Wenzel, M.: Efficient parallel programming in Poly/ML and Isabelle/ML. In: ACM SIGPLAN Workshop on Declarative Aspects of Multicore Programming (DAMP 2010). (2010)
[10] Moten, R.: Exploiting parallelism in interactive theorem provers. In Grundy, J., Newey, M.C., eds.: Theorem Proving in Higher Order Logics (TPHOLs 1998). Volume 1479 of LNCS., Springer (1998)
[11] Paulson, L.C.: Isabelle: the next 700 theorem provers. In Odifreddi, P., ed.: Logic and Computer Science. Academic Press (1990)
[12] Rager, D.L.: Adding parallelism capabilities in ACL2. In: Workshop on the ACL2 theorem prover and its applications (ACL2'06), ACM (2006)
[13] Rager, D.L.: Parallelizing an Interactive Theorem Prover: Functional Programming and Proofs with ACL2. Ph.d. dissertation, University of Texas at Austin (2012) http://www.cs.utexas.edu/~ragerdl/papers/dissertation/dissertation.pdf.
[14] Rager, D.L., Hunt, W.A., Kaufmann, M.: A parallelized theorem prover for a logic with parallel execution. In Blazy, S., Paulin-Mohring, C., Pichardie, D., eds.: Interactive Theorem Proving (ITP 2013). Volume ???? of LNCS., Springer (2013)
[15] Wenzel, M.: Parallel proof checking in Isabelle/Isar. In Dos Reis, G., Théry, L., eds.: ACM SIGSAM Workshop on Programming Languages for Mechanized Mathematics Systems (PLMMS 2009), ACM Digital Library (2009)
[16] Wenzel, M.: Isabelle/Isar — a generic framework for human-readable proof documents. In Matuszewski, R., Zalewska, A., eds.: From Insight to Proof — Festschrift in Honour of Andrzej Trybulec. Volume 10(23) of Studies in Logic, Grammar, and Rhetoric. University of Białystok (2007)
[17] Wenzel, M.: Isabelle/jEdit — a Prover IDE within the PIDE framework. In Jeuring, J., et al., eds.: Intelligent Computer Mathematics — 11th International Conference (CICM/MKM 2012). Volume 7362 of LNCS., Springer (2012)
[18] Wenzel, M.: READ-EVAL-PRINT in parallel and asynchronous proof-checking. In: User Interfaces for Theorem Provers (UITP 2012). EPTCS (2013)
[19] Wiedijk, F., ed.: The Seventeen Provers of the World. Volume 3600 of LNAI. Springer (2006)