A Parallelized Theorem Prover for Interactive Theorem Proving

David L. Rager, Warren A. Hunt Jr., and Matt Kaufmann
ragerdl@defthm.com, hunt@cs.utexas.edu,
kaufmann@cs.utexas.edu

July 26, 2013
Project Goals

- Add parallelism primitives to formal language
- Parallelize main ACL2 proof process
  - Provide proof debugging feedback more quickly
  - Reduce time required to replay proofs
Introduction to ACL2

- Functional programming language
- Theorem Prover is written in this programming language
- Automated theorem prover for first-order logic with induction
- Used by AMD, Centaur Technologies, IBM, Intel, Kestrel Institute, Rockwell Collins and other industrial, academic, and government sites

“... verified using Formal Methods techniques as specified by the EAL-7 level of the Common Criteria”
Warmup Example

- Goal (2 sec)
  - Subgoal 2 (7 sec)
    - Subgoal 2.2 (1 sec)
    - Subgoal 2.1 (3 sec)
  - Subgoal 1 (1 sec)
    - Subgoal 1' (5 sec)
    - Subgoal 1'' (2 sec)
Warmup Example

Time: 0.1 sec

Legend
- unstarted
- active
- pending
- finished

Goal (2 sec)

Subgoal 2 (7 sec)

Subgoal 2.2 (1 sec) Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)

Subgoal 1' (5 sec) Subgoal 1'' (2 sec)
Warmup Example

Time: 2.1 sec

Legend

unstarted  active  pending  finished
Warmup Example

Time: 9.1 sec

Legend
- unstarted
- active
- pending
- finished
Warmup Example

Time: 10.1 sec

Legend

- unstarted
- active
- pending
- finished

Goal (2 sec) ->
- Subgoal 2 (7 sec)
- Subgoal 1 (1 sec)

Subgoal 2.2 (1 sec)
Subgoal 2.1 (3 sec)
Subgoal 1' (5 sec)
Subgoal 1'' (2 sec)
Warmup Example

Time: 13.1 sec

Legend
- unstarted
- active
- pending
- finished
Warmup Example

Time: 14.1 sec

Legend: unstarted  active  pending  finished

Goal (2 sec)

- Subgoal 2 (7 sec)
  - Subgoal 2.2 (1 sec)
  - Subgoal 2.1 (3 sec)

- Subgoal 1 (1 sec)
  - Subgoal 1' (5 sec)
  - Subgoal 1'' (2 sec)
# Warmup Example

Time: 19.1 sec

- **Goal (2 sec)**
  - **Subgoal 2 (7 sec)**
    - **Subgoal 2.2 (1 sec)**
    - **Subgoal 2.1 (3 sec)**
  - **Subgoal 1 (1 sec)**
    - **Subgoal 1' (5 sec)**
    - **Subgoal 1'' (2 sec)**

*Legend*

- unstarted
- active
- pending
- finished
Warmup Example

Time: 21.0 sec

Legend
- unstarted
- active
- pending
- finished
Warmup Example

Time: 0.0 sec

Goal (2 sec)

Subgoal 2 (7 sec)

Subgoal 2.2 (1 sec)  Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)

Subgoal 1' (5 sec)

Subgoal 1'' (2 sec)

Legend

Can we make this proof go faster with parallel execution?
Can we make this proof go faster with parallel execution?
Warmup Example

Time: 2.1 sec

Can we make this proof go faster with parallel execution?
Can we make this proof go faster with parallel execution?
Warmup Example

Time: 8.1 sec

Goal (2 sec)

Subgoal 2 (7 sec)

Subgoal 2.2 (1 sec) Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)

Subgoal 1' (5 sec)

Subgoal 1'' (2 sec)

Legend

unstarted    active    pending    finished

• Can we make this proof go faster with parallel execution?
Warmup Example

Time: 9.1 sec

Goal (2 sec)

Subgoal 2 (7 sec)
  - Subgoal 2.2 (1 sec)
  - Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)
  - Subgoal 1' (5 sec)
  - Subgoal 1'' (2 sec)

Legend
  - unstarted
  - active
  - pending
  - finished

Can we make this proof go faster with parallel execution?
Warmup Example

Time: 10.1 sec

Goal (2 sec)

Subgoal 2 (7 sec)

Subgoal 2.2 (1 sec)  Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)

Subgoal 1' (5 sec)  Subgoal 1" (2 sec)

Legend  unstarted  active  pending  finished

Can we make this proof go faster with parallel execution?
Warmup Example

Time: 12.0 sec

Can we make this proof go faster with parallel execution?
Can we make this proof go faster with parallel execution?
Can first subgoal failure provide feedback sooner with parallel execution?
Warmup Example

Time: 13.1 sec

Goal (2 sec)

Subgoal 2 (7 sec)
  Subgoal 2.2 (1 sec)
  Subgoal 2.1 (3 sec)

Subgoal 1 (1 sec)
  Subgoal 1' (5 sec)
  Subgoal 1'' (2 sec)

Legend: unstarted, active, pending, finished

- Can we make this proof go faster with parallel execution?
- Can first subgoal failure provide feedback sooner with parallel execution?
Warmup Example

- Can we make this proof go faster with parallel execution?
- Can first subgoal failure provide feedback sooner with parallel execution?
Key Results

- Integrated parallelism primitives into the logic (and programming language)
- Many single-threaded features now thread-safe
- Use primitives to run theorem prover in parallel
- Created a robust implementation
  - 99.9% of the 80,000 theorem regression suite passes
  - 5.1x avg. speedup for 200 longest running theorems (32 cores)
  - Some theorems obtain a ≈25.7x speedup
  - At least a couple of users using subgoal-level parallelism on a daily basis
Related Work

- Many prior multi-threading primitives for Lisp
  - Multi-lisp, Parallel Lisp, Queue-based Multi-processing Lisp
  - Not embedded in logic of a theorem prover
- Process-level parallelism: ACL2, SiCoTHEO, Isabelle/HOL
- Theorem-level parallelism: Nqthm and Isabelle/HOL
- Proof-checking parallelism: Isabelle/HOL

We provide a stack of primitives embedded in the logic for an industrial-grade theorem prover, and we use these primitives to improve the *interactive* user's experience by *automatically* attempting the proofs of subgoals in parallel.
Outline

1. Introduction
2. Parallelism Primitives
3. Parallelizing ACL2
4. Evaluate Approach
5. Conclusion
Parallelism Primitives and Abstractions

Goal: Create Lisp and ACL2 primitives and abstractions necessary to parallelize the proof process

Results:
- Created multi-threading interface for Lisp
- Created futures library on top of this multi-threading interface
- Formalized speculative `spec-mv-let` primitive and implemented with futures
Fibonacci Example

(defun pfib (x)
  (if (or (zp x) (<= x 33))
      (fib x)
    (spec-mv-let
      (fib-x-1)
      (pfib (- x 1))
      (mv-let
         (fib-x-2)
         (pfib (- x 2))
         (if t
             (+ fib-x-1 fib-x-2)
             "Unreachable branch")))))

- Speedup of $\sim 7.4x$ on 8 cores (4 dual-core AMD Opteron 850s)
Interaction of Spec-mv-let, Threads, and Cores
Preparing ACL2

Preparing ACL2 for Parallel Execution

Goal: Make a single-threaded proof process thread-safe

Results:

- Disabled single-threaded features of ACL2’s main proof process
- Modified many of those features to be thread-safe
  - parallel output includes key components of serial output
- All but 11 (out of 3378) regression suite input files certify with parallelism
Preparing ACL2 for Parallel Execution

- **Goal:** Make a single-threaded proof process thread-safe
- **Results:**
  - Disabled single-threaded features of ACL2’s main proof process
  - Modified many of those features to be thread-safe
    - parallel output includes key components of serial output
  - All but 11 (out of 3378) regression suite input files certify with parallelism
    - computed hint that modifies global program state (2)
    - custom keyword hint that modifies global program state (1)
    - clause processor that modifies global program state (5)
    - profiling is not thread-safe (1)
    - infinitely recursive proof under parallel execution (2)
Executing ACL2 in Parallel

- Goal: Integrate parallel execution into the theorem proving process
- Results:
  - Parallelized the proof process with `spec-mv-let`, improving performance for non-trivial proofs
  - Feedback provided sooner than it would be with serial execution
One Refinement of Our Execution Strategy

- Problem: some parallel proofs caused machines to reboot
- Why does a user-level program cause a reboot?
- Behavior difficult to debug
One Refinement of Our Execution Strategy (cont’d)

- Original list-based approach requires $n$ threads while waiting on the last subgoal
- The threads associated with the $n^{th}$ subgoal could not return until $Subgoal < n-1>$ through $Subgoal 1$ finish their proofs
Hierarchical approach requires $\log(n)$ threads while waiting on the last subgoal.
Evaluate our Approach

- Goal: Investigate and articulate the benefits of parallel theorem proving
- Results:
  - Categorization scheme for proofs’ amenability to parallelism
  - Feedback provided sooner to the user
    - regardless of the number of CPU cores in the system
  - Reduced execution time
### Proof Categorization Scheme

<table>
<thead>
<tr>
<th>Category</th>
<th>Description</th>
<th>Early Feedback</th>
<th>Faster Execution</th>
<th>Count*</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Short</td>
<td>×</td>
<td>×</td>
<td>N/A</td>
</tr>
<tr>
<td>II</td>
<td>Long with late case-splits</td>
<td>×</td>
<td>×</td>
<td>1</td>
</tr>
<tr>
<td>III</td>
<td>Long with early case-splits and exactly one long path</td>
<td>✓</td>
<td>×</td>
<td>3</td>
</tr>
<tr>
<td>IV</td>
<td>Long with early case-splits and many long paths</td>
<td>✓</td>
<td>✓</td>
<td>21</td>
</tr>
</tbody>
</table>

*taken from the 25 longest running theorems
Survey of $\sim$80,000 ACL2 theorems

- Selected 200 longest running theorems
- Resulted in tuning implementation
- Many theorems in categories I, II, and III do not obtain significant speedup
- Many theorems in Category IV obtain significant speedup
- Challenge: sometimes the critical path lays dormant in the queue of subgoals that are to be executed in parallel.
Speedup for a 32-core without Hyper-threading

- Number of theorems with given speedup for 32-core Intel E5 (Sandy-bridge)
Defining Potential Speedup

- Potential speedup = total time / critical path time
- Critical path is 12 seconds
- Entire proof takes 21 seconds
- Potential speedup with an unlimited number of CPU cores is 21/12 \(\sim\) 1.75x
“Grading” with Potential Speedup

“Grading” Theorems Based upon Potential Speedup

- Grade approximates how close the actual speedup is to the potential speedup for a particular machine.
- Grade = actual speedup / min(core count, potential speedup)

  Example: a theorem that has a potential speedup of 100x, is executing on an 8-core machine, and has an actual speedup of 4x would receive a “grade” of 50%

  Example: a theorem that has a potential speedup of 2x, is executing on an 8-core machine, and has an actual speedup of 1.8x would receive a “grade” of 90%
“Grading” with Potential Speedup

“Grades” for a 32-core without Hyper-threading

- Number of theorems with given grade for 32-core Intel E5 (Sandy-bridge)

![Bar Chart]

- Number of theorems with given grade for 32-core Intel E5 (Sandy-bridge)

- 30.40%: 1
- 40.50%: 5
- 50.60%: 8
- 60.70%: 7
- 70.80%: 6
- 80.90%: 21
- 90-100%: 152
Lessons Learned

- Hierarchically breaking down work is better than a list-based approach
- Even code written in a functional style can have race conditions because side-effects are often hidden in the functional model
- Hyper-threading is as dangerous as it is useful
- Even proofs that do not experience much speedup can benefit from the breadth-first nature of parallel execution
- Subgoal-level parallelism is a useful level of granularity
- Recycling threads and dynamic allocation of parallelism resources are key
The Future

- Parallel proof capabilities will change users’ proof styles
- What can we try in parallel that we previously skipped because it was too outlandish for a single-core?
  - automated theory management
  - different proof libraries
  - multiple induction schemes
- What doors will open as we look beyond our goal of being backwards compatible with the regression suite?
Conclusion

- Results Achieved:
  - Added parallelism primitives to the theorem prover’s logic
  - Modified the theorem prover to be ready for parallel execution
  - Used our parallelism primitives to parallelize the execution of the theorem prover and obtain non-trivial speedup on many theorems
  - Provided key components of ACL2 output sooner than is available with serial execution

- In normal and regular use by users working on real projects
ACL2’s proof process (the *waterfall*)
Granularity of a subgoal
Life of Spec-mv-let
Life of a worker thread
Life of a piece of parallelism work
Full vs. resource-based waterfall parallelism
Benefits of hyper-threading
Critical path problem
Converting the ACL2 translator
Talking Points

ACL2’s Proof Process (the Waterfall)

- *The Waterfall* – simplification, induction, generalization, and other heuristics
- Proof is split into subgoals, which often require at least milliseconds to prove.
- Since the theorem prover is written in its own functional language, it is reasonable to introduce parallelism into ACL2’s proof process
- Spec-mv-let sufficiently general to insert into the code that implements the waterfall
Granularity of a subgoal

- Overhead of spec-mv-let is about 45µ
- Only 0.58% of the regression suite subgoals take less than 50µ
- Parallelizing at the subgoal level yields good granularity

<table>
<thead>
<tr>
<th>Range</th>
<th>Count of Subgoals</th>
<th>Percentage of Subgoals</th>
</tr>
</thead>
<tbody>
<tr>
<td>1µ to 50µ</td>
<td>6435</td>
<td>0.58%</td>
</tr>
<tr>
<td>51µ to 100µ</td>
<td>34174</td>
<td>3.05%</td>
</tr>
<tr>
<td>101µ to 150µ</td>
<td>25641</td>
<td>2.29%</td>
</tr>
<tr>
<td>151µ to 200µ</td>
<td>16211</td>
<td>1.45%</td>
</tr>
<tr>
<td>201µ to 250µ</td>
<td>12565</td>
<td>1.12%</td>
</tr>
<tr>
<td>251µ to 300µ</td>
<td>13171</td>
<td>1.18%</td>
</tr>
<tr>
<td>301µ to 350µ</td>
<td>12374</td>
<td>1.11%</td>
</tr>
<tr>
<td>351µ to 400µ</td>
<td>13976</td>
<td>1.25%</td>
</tr>
<tr>
<td>401µ to 450µ</td>
<td>17119</td>
<td>1.53%</td>
</tr>
<tr>
<td>451µ to 500µ</td>
<td>19341</td>
<td>1.73%</td>
</tr>
<tr>
<td>500+µ</td>
<td>947634</td>
<td>84.71%</td>
</tr>
</tbody>
</table>

Table: Number of subgoals with durations with the given time range
Life of Spec-mv-let

1. Spec-mv-let encountered
2. Creates task for speculative computation
3. Executes necessary computation
4. Test indicates speculative computation is useful?
   - Useful
     - Wait for speculative computation to complete
     - Execute true branch
     - Return
   - Unnecessary
     - Abort speculative computation (non-blocking)
     - Execute false branch
Life of a Worker Thread

1. **Thread Start**
   - Encounter another parallelism primitive and parallelize further?
     - No → **Idle**
     - Yes → **Active-R**
   - Obtain piece of work?
     - Yes → **Waiting-S**
     - No → **Active-S**

2. **Idle**
   - Obtain idle starting core
     - Yes → **Active-S**
     - No → **Waiting-R**

3. **Waiting-R**
   - Child finishes
     - Yes → **Pending**
     - No → **Waiting-R**

4. **Waiting-S**
   - Obtain idle resumptive core
     - Yes → **Active-S**
     - No → **Waiting-S**

5. **Active-R**
   - Obtain idle resumptive core
     - Yes → **Active-S**
     - No → **Waiting-R**

6. **Active-S**
   - Encounter parallelism primitive and parallelize further?
     - Yes → **Pending**
     - No → **Active-S**

7. **Pending**
   - Encounter parallelism primitive and parallelize further?
     - Yes → **Active-S**
     - No → **Pending**

8. **Thread Exit**
Life of a Piece of Parallelism Work

- Unassigned
  - Started
    - Encounter parallelism primitive and parallelize further?
      - Yes: Resumed
      - No: Proceed to next step
  - No: Proceed to next step
- No: Proceed to next step
- Yes: Proceed to next step
- Yes: Encounter another parallelism primitive and parallelize further?
  - Yes: Resumed
  - No: Finished
- No: Proceed to next step
Full vs. Resource-based Parallelism

- Resource-based was originally slightly better, then full, and now we’re back to resource-based.
- Resource management of the resource-based mode keeps the machine from reaching instability (and our user-level limits) while still providing efficient execution.
- Fixing the “backbone” problem lessens what used to be a dire need for resource-based parallelism but does not completely obviate it.
Benefits of Hyper-threading

- Take two theorems as a case study:
  
  **Theorem Ideal-8-way**
  
  - Obtains a speedup of $7.99\times$ on an eight core machine with no hyper-threading – a speedup of $1.00\times$ per core
  - Obtains a speedup of $3.92\times$ on a four core machine with two-way hyper-threading – a speedup of $0.98\times$ per core.
  - Hyper-threading is of no benefit to the proof of this theorem

  **JVM Theorem 2b**
  
  - Obtains a speedup of $6.50\times$ on an eight core machine with no hyper-threading – a speedup of $0.81\times$ per core
  - Obtains a speedup of $4.01\times$ on a four core machine with two-way hyper-threading – a speedup of $1.00\times$ per core.
  - Hyper-threading could provide a benefit of up to $23\%$ ($1.00/0.81-1$)

- In next slide, 8 theorems obtain speedup greater CPU core count

- We hypothesize that this is due to hyper-threading
Performance Statistics for a 4-core and Two-way Hyper-threaded Machine

- Number of theorems with given speedup for *dunnottar* (1 4-core Intel E31280)
Talking Points

Performance Statistics for a 4-core and Two-way Hyper-threaded Machine

- Number of theorems with given grade for dunnottar (1 4-core Intel E31280)
Illustration of the Critical Path Problem

- Theorem *Step2-marks-3marked-node-either-2-or-3-or-4*
  - Has a potential speedup of 6.74x
  - Requires 95.99 seconds to prove serially
  - Requires 24.11 seconds to prove in parallel on an 8-core machine, a speedup of 3.98x

What is happening?
- One of the longer subgoals, *Subgoal *1/5* does not start being proven until halfway through the proof of the general theorem
- *Subgoal *1/5* isn’t the most critical path, but when it starts that late, the proof is waiting for that subgoal to complete long after it has finished the other subgoals
- General Problem: The critical path is stuck idle in the buffer

- We could try to predict the critical path and prioritize it, but doing so requires a rework of the underlying parallelism implementation and is future work.
Possible Solutions to the Critical Path Problem

- How can we fix the critical path problem?
- Record time it takes to prove each subgoal
  - Requires a good “key” under which to store the duration
    - hash of the subgoal’s form?
- Use that information to prioritize that subgoal above other subgoals by moving that subgoal “up” in the parallelism work queue. This would be implemented at the level of futures, where we would use a priority queue instead of an array to store the futures that are to be executed.
  - An alternative implementation could change the order of the subgoals when we call the waterfall
    - Results in changing subgoal numbers
- Potential gotcha: once the critical path is prioritized, there can be other “second most critical” and “third most critical” paths.
ACL2 computed hints can be single-threaded
- First attempt disallowed all computed hints
- Some computed hints are actually thread-safe
- All computed hints must be run by the ACL2 translator
- Translator itself was single-threaded!
Converting the ACL2 Translator (cont’d)

- Made the translator thread-safe
  - Created and used a new mechanism for causing errors called context message pairs
- Translator now checks whether computed hint is thread-safe
- Provide mechanism to continue executing single-threaded computed hints

![Diagram showing the process flow: Computed hint (CH) → Translator → CH thread-safe? → Proceed or Error! → Hacks enabled? → Error! or Proceed]]