A Futures Library and Parallelism Abstractions for a Functional Subset of Lisp

David L. Rager  {ragerdl@cs.utexas.edu}
Warren A. Hunt, Jr.  {hunt@cs.utexas.edu}
Matt Kaufmann  {kaufmann@cs.utexas.edu}

The University of Texas at Austin

March 31, 2011
Motivation for our Talk

▶ Goals for today
  ▶ Present a library and ideas that may be of use in other systems
  ▶ Provide motivation for the further development of Lisp multi-threading capabilities and standards
  ▶ Gather feedback that results in a better implementation
Outline

Our Application: ACL2

Parallelism Primitives

Performance Results

Implementation Improvements since ILC 2009

Related Work

Conclusion
Outline

Our Application: ACL2
   Description
   Proof Process

Parallelism Primitives

Performance Results

Implementation Improvements since ILC 2009

Related Work

Conclusion
Description of ACL2

- Functional programming language (contains car, cons, assoc, etc.)
- ACL2 Theorem Prover is written in this ACL2 programming language
- Semi-automatic theorem prover for first-order logic with induction
- Used by AMD, IBM, Centaur Technologies, and Rockwell Collins to model and verify parts of their chips; also used at other industrial, academic, and government sites

“verified using Formal Methods techniques as specified by the EAL-7 level of the Common Criteria”
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.
- Our five parallelism primitives are created specifically with our application and code’s shape in mind.
Outline

Our Application: ACL2

Parallelism Primitives
  Futures
  Spec-mv-let
  Plet+

Performance Results

Implementation Improvements since ILC 2009

Related Work

Conclusion
Futures

- Goal – provide an efficient mechanism for parallel evaluation in Lisp
- Future – similar to an identity macro, except it returns a data structure, such that when `future-read` is applied to it, returns the result of evaluating `future`’s argument
  - Key convenience – `future`’s argument is often evaluated in another thread
- `Future-read` – applied to the data structure returned by `future` to obtain an computation’s evaluation result
- `Future-abort` – aborts the evaluation of a future (a.k.a. *early termination*)
- Example: `(future-read (future 3)) \Rightarrow 3`

---

1Halstead, “Implementation of Multilisp: Lisp on a Microprocessor”, 1984
Futures Example

(defun pfib (x)
  (if (< x 33)
      (fib x)
      (let ((a (future (pfib (- x 1))))
            (b (future (pfib (- x 2)))))
        (+ (future-read a)
            (future-read b))))

- Speedup of 7.5-8x on 8-core system for (pfib 45)
Spec-mv-let

- Goal – provide an efficient mechanism for parallel evaluation of the ACL2 theorem prover
- Short for Speculative Multiple Value Let (mv-let)
- Mv-let is ACL2’s version of multiple-value-bind
Spec-mv-let General Form

(spec-mv-let
  (v1 ... vn) ; bind distinct variables
  <spec-form> ; evaluate speculatively; return n values
(mv-let
  (w1 ... wk) ; bind distinct variables
  <eager-form> ; evaluate eagerly
  (if <test-form> ; ignore <spec> if true
    ; (does not mention v1 ... vn)
    <abort-form> ; does not mention v1 ... vn
    <normal-form>))) ; may mention v1 ... vn

- In our application, <eager-form> represents performing the proof process on the first proof subgoal, while <spec-form> represents speculatively proving the remaining subgoals.
- By calling the function that uses spec-mv-let recursively, we parallelize ACL2’s proof process at the subgoal level.
(defun pfib (x)
  (if (< x 33)
      (fib x)
      (spec-mv-let
       (a)
       (pfib (- x 2))
       (mv-let
        (b)
        (pfib (- x 1))
        (if nil
            "speculative result is always needed"
            (+ a b)))))

▶ Speedup of 7.5-8x on 8-core system for (pfib 45)
Goal – provide a more general mechanism for parallel evaluation in ACL2

Similar to `let` but has three additional features:

1. Can evaluate its bindings concurrently (as with `plet` from ILC 2009)
2. Allows the programmer to bind not just single values but also multiple values
3. Supports speculative evaluation, blocking only when a binding’s value is needed in the body of the form

Thus far used in small examples, but we plan to improve it for use in the ACL2 proof process and for ACL2 programmers
(defun pfib (x)
  (if (< x 33)
      (fib x)
      (plet+ ((a (pfib (- x 1)))
              (b (pfib (- x 2))))
              (with-vars (a b)
                         (+ a b))))))

Speedup of 7.5-8x on 8-core system for (pfib 45)
Outline

Our Application: ACL2

Parallelism Primitives

Performance Results
  Testing Parameters
  Futures, Spec-mv-let, and Plet+
  ACL2 Proofs
    Effects of Garbage Collection
    Other ACL2 Theorems

Implementation Improvements since ILC 2009

Related Work

Conclusion
Testing Parameters

- 8 core system
- 64 bit CCL results only, with EGC disabled/enabled and a varied GC threshold
- Minimum, maximum, and average wall clock times for ten consecutive executions of each test
Futures, Spec-mv-let, and Plet+

Figure: Performance of Parallelism Primitives in the Fibonacci Function

<table>
<thead>
<tr>
<th>Case</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>40.06</td>
<td>40.21</td>
<td>40.08</td>
<td></td>
</tr>
<tr>
<td>Futurized</td>
<td>5.15</td>
<td>5.78</td>
<td>5.26</td>
<td>7.62</td>
</tr>
<tr>
<td>Spec-mv-let</td>
<td>5.13</td>
<td>5.22</td>
<td>5.17</td>
<td>7.75</td>
</tr>
<tr>
<td>Plet+</td>
<td>5.08</td>
<td>5.18</td>
<td>5.12</td>
<td>7.82</td>
</tr>
</tbody>
</table>

- Speedup ranges from 6.95 to 7.88, with the reported averages
- Large variance is caused by the underlying runtime systems
- Ephemeral Garbage Collection was disabled and we had a high GC threshold of 16 gigabytes
- Called the garbage collector before each test and manually checked that it did not run during that test
- Therefore the variance is not caused by garbage collection
ACL2 Proofs

- Currently use primitive `spec-mv-let`
- Garbage collection plays a large role in the performance of our proofs
  - Analyze the effects of GC with theorem `JVM-2A`
  - Show speedup of other theorems under the optimal GC configuration
Effects of Garbage Collection

- Two parameters:
  - Ephemeral Garbage Collector (enabled vs. disabled)
  - Garbage Collection threshold (default vs. 16 gigabytes)
Effects of Garbage Collection Results

**Figure:** Performance of Theorem JVM-2A with Varying GC Configurations

<table>
<thead>
<tr>
<th>EGC &amp; Threshold</th>
<th>Case</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>on, default</td>
<td>serial</td>
<td>245.52</td>
<td>246.99</td>
<td>246.79</td>
<td>0.60</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>372.54</td>
<td>482.62</td>
<td>413.42</td>
<td></td>
</tr>
<tr>
<td>on, high</td>
<td>serial</td>
<td>245.38</td>
<td>247.09</td>
<td>246.90</td>
<td>0.58</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>377.91</td>
<td>524.78</td>
<td>422.20</td>
<td></td>
</tr>
<tr>
<td>off, default</td>
<td>serial</td>
<td>291.57</td>
<td>292.14</td>
<td>291.97</td>
<td>2.54</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>110.57</td>
<td>117.17</td>
<td>114.77</td>
<td></td>
</tr>
<tr>
<td>off, high</td>
<td>serial</td>
<td>229.79</td>
<td>242.40</td>
<td>231.14</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>34.42</td>
<td>39.42</td>
<td>35.51</td>
<td></td>
</tr>
</tbody>
</table>
Effects of Garbage Collection Analysis

- Serial evaluation benefits from the EGC in low-memory environments
- Both serial and parallel evaluation benefit from disabling the EGC in high-memory environments
- Both serial and parallel evaluation are fastest with the EGC disabled and a high GC threshold
- We therefore run all of our application’s tests with the EGC disabled and a high GC threshold.
Reflection upon Effects of Garbage Collection

- The community has recognized multi-core computing as being pervasive
- The community has developed well-established multi-threading libraries (based off pthreads)
- Until the garbage collectors are parallelized, the use of these multi-threading libraries is greatly weakened in any GC-intense application
Other ACL2 Theorems

Four Theorems:

- *Embarrassingly Parallel* – Designed by us to show the ideal speedup of our application
- *JVM-2A* – About a JVM model constructed in ACL2
- *Measure 2* and *Measure 3* – Aid in proving the termination of Takeuchi’s Tarai function
## Other ACL2 Theorems Results

**Figure:** Performance of ACL2 Proofs with the EGC Disabled and a High GC Threshold

<table>
<thead>
<tr>
<th>Proof</th>
<th>Case</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Embarrassing</td>
<td>serial</td>
<td>36.49</td>
<td>36.53</td>
<td>36.50</td>
<td>7.93</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>4.58</td>
<td>4.61</td>
<td>4.60</td>
<td></td>
</tr>
<tr>
<td>JVM-2A</td>
<td>serial</td>
<td>229.79</td>
<td>242.40</td>
<td>231.14</td>
<td>6.51</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>34.42</td>
<td>39.42</td>
<td>35.51</td>
<td></td>
</tr>
<tr>
<td>Measure-2</td>
<td>serial</td>
<td>175.99</td>
<td>179.93</td>
<td>176.53</td>
<td>3.53</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>47.07</td>
<td>53.71</td>
<td>50.01</td>
<td></td>
</tr>
<tr>
<td>Measure-3</td>
<td>serial</td>
<td>86.63</td>
<td>86.85</td>
<td>86.73</td>
<td>3.48</td>
</tr>
<tr>
<td></td>
<td>par</td>
<td>24.24</td>
<td>25.36</td>
<td>24.90</td>
<td></td>
</tr>
</tbody>
</table>
Outline

Our Application: ACL2

Parallelism Primitives

Performance Results

Implementation Improvements since ILC 2009
  Use of Arrays and Atomic Increments
  Early Termination of Futures

Related Work

Conclusion
Use of Arrays and Atomic Increments

- 2009 version of our library used a shared work-queue
- Pushed pieces of parallelism onto the back of the work-queue
- FIFO ordering
- Required locking the work-queue while performing the nconc or popping from the work-queue
- Instead, we now use a shared array
- Pieces of parallelism work are added and chosen for evaluation using atomic increments
  - Now make heavy use of atomic increments and decrements in CCL
  - Lock-free
Early Termination of Futures

(defun mistake ()
    (future-abort (future (count-down 1000000000))))

(time
    (dotimes (i 100000)
        (mistake)))

- Count-down is designed to burn CPU time, and the above call of count-down takes about 5 seconds
- Calling mistake, as above, should take 100,000 * 5 seconds
- Takes about 6 seconds
- We have a new early termination mechanism, made for futures, which is documented in the file futures-mt.lisp
  - 72,000 evaluations abort by reading a flag, checked before starting
  - 28,000 evaluations abort by being thrown
  - Lock-free
Outline

Our Application: ACL2

Parallelism Primitives

Performance Results

Implementation Improvements since ILC 2009

Related Work

Conclusion
Related Work

- 80s Contributions: Multilisp, Parallel Lisp, futures, etc.
- Haverbeke’s PCall library
- Sedach’s Eager Future’s library
- Bordeaux Threads project
- Isabelle theorem prover
- Herzeel and Costanza’s use of recursion in parallelizing Scheme
Outline

Our Application: ACL2

Parallelism Primitives

Performance Results

Implementation Improvements since ILC 2009

Related Work

Conclusion
Conclusion

- Provide futures, spec-mv-let, and plet+ primitives
- Used these primitives to parallelize the key ACL2 proof process
- Garbage collection is a major bottleneck in the parallelized performance of applications with large amounts of garbage, but even so we were able to get 3.5x-7.9x speedup on proofs with lots of subgoals
Obtaining Our Library

- Library available as part of an experimental branch of ACL2
- We are happy to provide a tarball of this branch upon request, which implements these parallelism primitives for both CCL and SBCL