Parallelizing an Interactive Theorem Prover
Functional Programming and Proofs with ACL2

David L. Rager  ragerdl@cs.utexas.edu

The University of Texas at Austin

August 20, 2012
Project Goals

- Add parallelism primitives to formal language
- Parallelize large program written in that formal language
  - 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, and Rockwell Collins, and used at other industrial, academic, and government sites

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

Time: 0.1 sec

Legend
- unstarted
- active
- pending
- finished
Warmup Example

Time: 2.1 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: 9.1 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: 10.1 sec

Legend
- unstarted
- active
- pending
- finished
Warmup Example

Time: 13.1 sec

Legend  unstarted  active  pending  finished
Warmup Example

Time: 14.1 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: 19.1 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: 21.0 sec

Legend:
- unstarted
- active
- pending
- finished

Graph:
- 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

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

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

Time: 3.1 sec

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

Time: 8.1 sec

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

Time: 10.1 sec

Legend: unstarted, active, pending, finished

- 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)

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

Time: 12.0 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 first subgoal failure provide feedback sooner with parallel execution?
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)

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

Time: 2.1 sec

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

1. Create primitives in a formal logic necessary to introduce parallel execution into applicative programs, while preserving these programs’ amenability to verification

2. Prepare the theorem prover for parallel execution, paying attention to interactive issues, and then integrate parallel execution into its proof process

3. Evaluate our approach
Key Results

- Integrated parallelism primitives into the logic (and programming language)
- Many single-threaded features now thread-safe
- Use `spec-mv-let` to run theorem prover in parallel
- Created a robust implementation

  - 99.9% of the 80,000 theorem regression suite certifies
  - 3.7x avg. speedup for 200 longest running theorems (20 cores)
  - Some theorems obtain a $\sim 13x$ speedup
  - Theorems designed for parallel execution obtain $\sim 19x$ speedup
  - Several users excited about parallel proof
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: ACL2 and Isabelle/HOL
- Proof-checking parallelism: Isabelle/HOL
- No one has automated interactive subgoal level parallelism

- 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.
Overview

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 `spec-mv-let` primitive and implemented with futures
Spec-mv-let General Form

```
(spec-mv-let
  (v1 ... vn)
  <spec>
  (mv-let
    (w1 ... wm)
    <eager>
    (if <test>
      <typical-case>
      <abort-case>))
```

- "speculative" call
- "necessary" call
- work queue
- current thread
- chance to abort early
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
       (valid2 count2)
       (integer-treep (cdr tree))
       (mv-let (valid1 count1)
                 (integer-treep (car tree))
                 (if valid1
                     (mv valid2 (+ count1 count2))
                     (mv nil 0)))))))
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
        (valid2 count2)
        (integer-treep (cdr tree))
        (mv-let (valid1 count1)
          (integer-treep (car tree))
          (if valid1
              (mv valid2 (+ count1 count2))
              (mv nil 0))))))

Legend:
- necessary
- speculative

Base case:
- work queue
- current thread

Speculative call:
- chance to abort early

Necessary call:
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
       (valid2 count2)
       (integer-treep (cdr tree))
       (mv-let (valid1 count1)
        (integer-treep (car tree))
        (if valid1
            (mv valid2 (+ count1 count2))
            (mv nil 0))))))

Legend
- necessary
- speculative

Base case
Speculative call
Necessary call
Chance to abort early
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
        (valid2 count2)
        (integer-treep (cdr tree))
        (mv-let (valid1 count1)
          (integer-treep (car tree))
          (if valid1
              (mv valid2 (+ count1 count2))
              (mv nil 0))))))
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
       (valid2 count2)
       (integer-treep (cdr tree))
       (mv-let (valid1 count1)
              (integer-treep (car tree))
              (if valid1
                  (mv valid2 (+ count1 count2))
                  (mv nil 0))))))
General Form and Examples

Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
        (valid2 count2)
        (integer-treep (cdr tree))
        (mv-let (valid1 count1)
          (integer-treep (car tree))
          (if valid1
              (mv valid2 (+ count1 count2))
              (mv nil 0))))))

Legend

necessary

speculative

base case

"speculative" call

"necessary" call

work queue

current thread

chance to abort early
Recursive Integer Tree Example

```lisp
(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
        (valid2 count2)
        (integer-treep (cdr tree))
        (mv-let (valid1 count1)
          (integer-treep (car tree))
          (if valid1
              (mv valid2 (+ count1 count2))
              (mv nil 0)))))))
```

Legend:
- **necessary**
- **speculative**

Base case: (atom tree)

"speculative" call:
- (integer-treep (cdr tree))

"necessary" call:
- (integer-treep (car tree))
- (mv (integerp tree) 1)

Work queue:
- (spec-mv-let (valid2 count2))

Current thread:
- (mv-let (valid1 count1))
- (if valid1 (mv valid2 (+ count1 count2)) (mv nil 0)))
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
       (valid2 count2)
       (integer-treep (cdr tree))
       (mv-let (valid1 count1)
                (integer-treep (car tree))
                (if valid1
                    (mv valid2 (+ count1 count2))
                    (mv nil 0))))))
Recursive Integer Tree Example

(defun integer-treep (tree)
  (if (atom tree)
      (mv (integerp tree) 1)
      (spec-mv-let
        (valid2 count2)
        (integer-treep (cdr tree))
        (mv-let (valid1 count1)
          (integer-treep (car tree))
          (if valid1
              (mv valid2 (+ count1 count2))
              (mv nil 0)))))

Legend
- necessary
- speculative

Base case
-speculative call
-necessary call
-chance to abort early

Work queue
Current thread
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"))))

- not worth parallelizing
- "speculative" recursive call
- "necessary" recursive call
- chance to abort early

- Speedup of \(~7.4x\) on 8 cores (4 dual-core AMD Opteron 850s)
Interaction of Primitives, Threads, and Cores

Interaction of Spec-mv-let, Threads, and Cores

Legend
- empty
- unassigned
- active
- pending

Worker threads

CPU cores

Thread

spec-mv-let

Work queue

+ task
**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? (Flow splits here)
   - **Useful**
     - Wait for speculative computation to complete
     - Execute true branch
     - Return
   - **Unnecessary**
     - Abort speculative computation (non-blocking)
     - Execute false branch
Preparing ACL2 for Parallel Execution

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

- **Goal:** Make a single-threaded proof process thread-safe
- **Results:**
  - Disabled single-threaded features of ACL2’s proof process
  - Modified many of those features to be thread-safe
    - parallel output includes key components of serial output
  - All but 11 (currently out of 3378) regression suite input files certify using 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)
Converting the ACL2 Translator

- 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

```
Composed hint (CH) → Translator → CH thread-safe?
  Yes → Proceed
  No → Hacks enabled?
  No → Error!
  Yes → Yess
```
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.
Overview

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
<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 \(\sim80,000\) 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 an 8-core without Hyper-threading**

- Number of theorems with given speedup for *lhug* (4 dual-core AMD Opteron 850s)
Speedup for a 20-core with Two-way Hyper-threading

- Number of theorems with given speedup for *glamdring* (2 10-core Intel E7-2870s)
“Grading” with Potential Speedup

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 \( \frac{21}{12} \approx 1.75x \)

\[
\begin{align*}
\text{Goal (2 sec)} \\
\text{Subgoal 2 (7 sec)} & \quad \text{Subgoal 1 (1 sec)} \\
& \quad \text{Subgoal 2.2 (1 sec)} \quad \text{Subgoal 2.1 (3 sec)} \\
& \quad \text{Subgoal 1'} (5 sec) \quad \text{Subgoal 1'' (2 sec)}
\end{align*}
\]
“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 / \( \min(\text{core count, potential speedup}) \)

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 an 8-core without Hyper-threading

- Number of theorems with given grade for *lhug*
  (4 dual-core AMD Opteron 850s)

![Chart showing number of theorems with given grades](chart.png)
“Grading” with Potential Speedup

“Grades” for a 20-core with Two-way Hyper-threading

- Number of theorems with given grade for *glamdring* (2 10-core Intel E7-2870s)
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
  - :or hints
  - multiple induction schemes
  - reverting to prove by induction while concurrently continuing the current proof attempt
- What can we model and run efficiently because of our work?
  - state-based approach to modeling pthread-like primitives
  - efficient execution because we now have a native implementation on many Lisps
- What doors will open as we look beyond our goal of being backwards compatible with the regression suite?
Future Implementation Work

- Prioritize critical proof paths
- Improve support for remaining single-threaded features
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
- Articulated the ways that subgoal-level parallelism can benefit users of interactive theorem provers

- In normal and regular use by users working on real projects
Acknowledgements

- Dissertation Committee: James C. Browne, Matt Kaufmann, J Strother Moore, Jun Sawada, Emmett Witchel
- ACL2 Integration: Matt Kaufmann
- Colleagues: Gary Byers, Pascal Constanza, Jared Davis, Shilpi Goel, Marijn Heule, Robert Krug, Sung Jun Lim, Martin Simmons, Sol Swords, Nathan Wetzler, Bill Young
- Family and friends
Additional Talking Points

- ACL2’s proof process (the waterfall)
- Granularity of a subgoal
- 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
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 $45\mu$
- Only 0.58% of the regression suite subgoals take less than $50\mu$
- 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 a Worker Thread

- **Thread Start**
  - Yes: Obtain piece of work?
  - No: Encounter another parallelism primitive and parallelize further?

- **Idle**
  - Yes: Obtain piece of work?
  - No: Encounter another parallelism primitive and parallelize further?

- **Active-R**
  - Yes: Obtain idle resumptive core
  - No: Child finishes

- **Waiting-R**

- **Active-S**
  - Yes: Obtain idle starting core

- **Waiting-S**
  - Yes: Obtain piece of work?
  - No: Encounter parallelism primitive and parallelize further?

- **Pending**

- **Thread Exit**
Life of a Piece of Parallelism Work

1. Unassigned
   - No

2. Started
   - Encounter parallelism primitive and parallelize further?
     - No
     - Finished
     - Yes
     - Pending

3. Pending
   - No
   - Yes
   - Resumed

4. Resumed
   - No
   - Yes
   - Encountered another parallelism primitive and parallelize further?
     - No
     - Finished
     - Yes
     - Started
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 7.99x on an eight core machine with no hyper-threading – a speedup of 1.00x per core
    - Obtains a speedup of 3.92x on a four core machine with two-way hyper-threading – a speedup of 0.98x per core.
    - Hyper-threading is of no benefit to the proof of this theorem
  - **JVM Theorem 2b**
    - Obtains a speedup of 6.50x on an eight core machine with no hyper-threading – a speedup of 0.81x per core
    - Obtains a speedup of 4.01x on a four core machine with two-way hyper-threading – a speedup of 1.00x 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)
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.