Behind the Scenes of PACL2

April 23, 2007

David L. Rager

Outline

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - Early termination implementation
  - Determining resource availability
- Work Accomplishments
- Future Work

Parallelism Primitives

- PCALL
- PLET
- PAND
- POR

- When parallelism resources are available, parallel evaluation is introduced. When resources are unavailable, these primitives execute a serial equivalent

PCALL

- Logically the identity macro
- May evaluate the arguments to a function call in parallel and applies the function to the results of the evaluation

- Example form:

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (pcall (binary-+ (pfib (- x 1))
                            (pfib (- x 2)))))))
```

PLET

- Logically equivalent to LET
- May evaluate variable bindings in parallel and applies a closure created from the body of the plet to the results of the binding evaluations
- Example form:

```
(defun pfib (x)
  (cond ((or (zp x) (<= x 0)) 0)
        ((= x 1) 1)
        (t (plet ((fibx-1 (pfib (- x 1)))
                  (fibx-2 (pfib (- x 2))))
                 (+ fibx-1 fibx-2)))))
```

POR

- Logically similar to OR
- May evaluate its arguments in parallel, evaluates their disjunction, and returns a Boolean result
  - (or (atom x) (car x)) is guard-verifiable
  - (por (atom x) (car x)) is not guard-verifiable
  - Boolean result to maintain consistency between executions
- Example form:

```
(defun invalid-tree (x)
  (if (atom x)
      (invalid-tip x)
    (por (invalid-tree (car x))
         (invalid-tree (cdr x)))))
```

PAND

- Logically similar to AND
- May evaluate its arguments in parallel, evaluates their conjunction, and returns a Boolean result
  - (and (consp x) (car x)) is guard-verifiable
  - (pand (consp x) (car x)) is not guard-verifiable
  - Boolean result to be consistent with POR

- Example form:

```
(defun valid-tree (x)
  (if (atom x)
      (valid-tip x)
    (pand (valid-tree (car x))
          (valid-tree (cdr x)))))
```

PAND/POR

- Remember that PAND and POR have early termination conditions
  - (pand (car x) (cdr x)) may never evaluate (car x)

Outline

- Parallelism Primitives
- Evaluation Strategy
  - *Producer/Consumer roles and implementation*
  - Early termination implementation
  - Determining resource availability
- Work Accomplishments
- Future Work

Work Producer/Consumer Implementation

- Who is the first work producer?
  - You!
- How do you become a work producer?
  - Use a parallelism primitive

Work Producer/Consumer Implementation

- Brainstorm ideas on how we could implement a work producer
  - The producer could evaluate the work itself
    - (plet ((x (foo n)) (y (bar n))) (+ x y))
      - The producer itself could evaluate closures that represent (foo n) and (bar n)
      - Does this result in any speedup?
        - No. (of course not)

Work Producer/Consumer Implementation

- Brainstorm ideas on how we could implement a work producer
  - The producer could evaluate the work itself
    - A piece of work would contain
      - A closure
      - Result array (place to store the result)
      - An index into the array

Work Producer/Consumer Implementation

- Brainstorm ideas on how we could implement a work producer
  - The producer could spawn threads
    - Create the threads to evaluate the work you want done in parallel
      - OpenMCL allows you to tell a starting thread which function to run
      - This function could evaluate a piece of work, store the result in a shared variable, and signal a semaphore to let the producer know when it's done.
    - Once all the child work is finished, the producer can apply the specified function to the results and return the answer

Work Producer/Consumer Implementation

- Brainstorm ideas on how we could implement a work producer
  - The producer could spawn threads
    - A piece of parallelism work would contain
      - Closure
      - Result array (place to store the result)
      - An index into the array
      - Semaphore (for signalling the parent when the result is stored)

Work Producer/Consumer Implementation

- The producer could spawn threads
  - Advantages:
    - The spawned threads expire after finishing evaluating their work, freeing OS resources
  - Disadvantages:
    - No buffering of work – parallelism is either actively introduced and evaluated or the producer decides to evaluate serially
    - Creating threads takes time. Spawning a thread for each parallel computation results in a 10x slowdown (see draft of thesis for details).

Work Producer/Consumer Implementation

- Brainstorm ideas on how we could implement a work producer
  - The producer could push the work on a global work queue, and other threads could be standing by to evaluate the work.
  - Requires someone to make sure there are an appropriate number of idle "worker" threads
    - The producer can handle that much!
  - Requires someone to say when more work has been added to the system
    - The producer can handle that much!

Work Producer/Consumer Implementation

- Worker Threads
  - The producer could push its work on a work queue
    - Advantages
      - OpenMCL allows you to tell a starting thread which function to run
      - This function could wait until work is added to the system, calculate (foo n), store the result somewhere, signal a semaphore to let us know when it's done, and rewait.
      - Don't pay 10x overhead for spawning new threads
      - Work is buffered
    - Disadvantages
      - Worker threads standing-by should be given a way to expire
        - We do this for OpenMCL via a timed wait and throw/catch pair described in the code and paper

Work Producer Implementation

- A work producer is responsible for:
  - Putting pieces of parallelism work on the work queue
  - Creating the worker threads if necessary, and
  - Signaling the correct variables to ensure that this work gets evaluated
- A piece of parallelism work contains:
  - Closure
  - Result array (place to store the result)
  - An index into the array
  - Semaphore (for signalling the parent when the result is stored)
  - Terminate early function (for terminating early)
  - Thread-array (also for terminating early, more later)

Work Consumer Implementation

- Worker threads begin as work consumers
- Responsible for:
  - Taking work off the work queue
  - Evaluating it
  - Terminating irrelevant related work
  - Saving the result
- Messy implementation mechanisms
  - Unwind-protects with interrupts disabled during cleanup
  - Special/thread-local variables
  - Two different throw/catch tag pairs
  - Optimistic CPU core grabbing
  - “Forever” loops

Work Consumer Implementation

- How many worker threads do we need?
  - A worker thread is in one of four states:
    - Idle (waiting for work and CPU core)
    - Started (it has a piece of work and is consuming CPU cycles)
    - Pending (it has become a producer and is now waiting for child work to finish evaluating)
    - Resumed (its child work has finished evaluating)
  - If p is the number of CPU cores, then we want the sum of the number of idle and active (started + resumed) threads to be $2p$
  - Any time a producer adds a piece of parallelism work to the work queue, it spawns a number of worker threads, such that the number of idle and active threads is $2p$

Outline

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - *Early termination implementation*
  - Determining resource availability
- Work Accomplishments
- Future Work

Early Termination Implementation

- Three ways for work to be terminated:
  - A sibling decides the result is irrelevant
  - A sibling of the parent decides the result is irrelevant
  - The user decides that the result is irrelevant
    - The code for this case is the exact same as the second case, so its further discussion is omitted

Early Termination Implementation

- A sibling decides that the result is irrelevant
  - This involves the sibling:
    - Removing all its siblings' work from the work queue and
    - Interrupting all its siblings with a function that aborts work evaluation (this is one of those throw/catch pairs I mentioned earlier).
  - The abortion process is done in a certain order and uses low-level variables to ensure that work that's between the "off the work-queue" and "active" stages is always aborted.

Early Termination Implementation

- A parent decides that the result is irrelevant
  - This involves the parent:
    - Removing all its *child* work from the work queue and
    - Interrupting all its children with a function that aborts parallelism work evaluation (this is the same throw/catch pairs I mentioned in the previous slide).
  - The abortion process is done in a certain order and uses low-level variables to ensure that work that's between the "off the work-queue" and "active" stages is always aborted.

Outline

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - Early termination implementation
  - *Determining resource availability*
- Work Accomplishments
- Future Work

Determining Resource Availability

- Two types of resources:
  - Threads
    - Need to limit the number of threads, so that the OS doesn't blow us up
    - Goal: max-out total count of threads around 50 (or some other semi-arbitrary number)
  - CPU Cores
    - Need to keep CPU cores busy
    - Need to minimize context-switching
    - Goal: Keep total number of active threads equal to the number of CPU Cores (8 in this presentation)

Determining Resource Availability

- Threads
  - Need to limit the number of threads, so that the OS doesn't blow us up
    - Solution: The heuristics that the producer uses to determine whether to parallelize computation will always result in serial evaluation if there are more than 50 pieces of work in the system at any point
    - Why do we need to limit the work? Why not just limit the number of threads spawned?
      - All work that enters the parallelism system must be dealt with somehow (either via evaluation or early termination). Since the only way to evaluate work is with a worker thread, if the work is added, the worker thread must be spawned. We therefore must limit the work at the root, where it is added.

Determining Resource Availability

- CPU Cores
  - Need to keep CPU cores busy
    - Supposing there are *p* CPU cores, there will always be around *p* pieces of work on the work queue.
    - Note that once a worker thread grabs a piece of work, that it is not on the work queue.
    - The work queue, therefore acts as a buffer that allows worker threads to immediately grab another piece of work once they finish evaluating their current piece

Determining Resource Availability

- CPU Cores
  - Need to minimize context-switching
    - A worker thread is in one of four states:
      - Idle (also referred to as "standing by" earlier in this presentation)
      - Started (it has a piece of work and is consuming CPU cycles)
      - Pending (it has become a consumer and is now waiting for child work to finish evaluating)
      - Resumed (its child work has finished evaluating)
    - Note that unless the worker thread becomes a parallelism producer, a worker thread's lifecycle only consists of the first two phases
    - A worker thread is considered active if it is in the started or resumed state
    - Supposing there are *p* CPU cores, there should be *p* active threads at any moment. But wait…

Determining Resource Availability

- **CPU Cores**
  - Need to minimize context-switching
    - Supposing there are *p* CPU cores, there should be *p* active threads at any moment. But wait…
    - What happens when there are eight *started* workers, and a worker needs to *resume*? Can the *resuming* worker pre-empt one of the *started* workers somehow? Does it matter?
      - Often the closure being applied to results of evaluating arguments in parallel is faster than the argument evaluations
      - It would therefore be nice to able to get this application "out of the way" and free the *resuming* worker thread for further parallelism work evaluation

Determining Resource Availability

- CPU Cores
  - Need to minimize context-switching
    - What happens when there are eight *started* workers, and a worker needs to *resume*? Can the *resuming* worker pre-empt one of the *started* workers somehow?
      - It is complicated to interrupt a *started* worker and tell it to pause until the *resumed* thread is done. So we don't pre-empt the *started* thread.
    - Instead, we let the *resumed* thread to run.
      - But how? Is there a limit to the number of *resumed* threads?
        - Yes, let's explain the implementation

Determining Resource Availability

- CPU Cores
  - Implementation of *started* and *resuming* thread CPU core sharing
    - Before a *starting* worker thread can actually start, it will check a shared variable, namely *idle-core-count*, to ensure that it is > 0. This count is decremented once the thread passes this check.
      - This ensures that a *starting* thread will never consume CPU cycles, unless there are cycles to spare
    - Before a resuming worker thread can actually resume, it will check the same shared variable, namely *idle-core-count*, to ensure that it is > (-p).
      - This ensures that a *resuming thread* will never be context-switching with more than one other thread.
      - We hope the OS can handle scheduling two threads on one CPU core.

Tying Resource Availability to the Producer

- A producer will only add work to the work queue if the following three conditions are true:
  - Parallelism is enabled
  - The total amount of active work and work in the work queue and is less than twice the number of CPU cores
  - The total amount of work already in the system is less than a somewhat arbitrary limit (50 in the previous slides).
- Notice that there is nothing about the number of idle CPU cores in this heuristic, just that there is enough work in the work queue. The work consumer handles the idle CPU core problem.

Tying Resource Availability to the Consumer

- A worker thread will only begin work evaluation if the following two conditions are true:
  - There is an idle CPU core
  - There is a piece of work to grab
- Notice that there is nothing about the number of worker threads. The work producer handles this problem.

Outline

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - Early termination implementation
  - Determining resource availability
- *Work Accomplishments*
- Future Work

Work Accomplishments

- Separation of work producer and work producer roles (in both explanation and code), even though it could be a mess since worker threads can be both producers and consumers
- Noticeable speedup on problems with relatively little GC.
- Matches and beats speedup of lazy if evaluation by using early termination
- Provides a logical foundation for a distributed ACL2
- A side-effect of the work is an interface for using LISP-level parallelism primitives in LISP
  - signal-semaphore, signal-condition-variable, make-lock, etc…

Outline

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - Early termination implementation
  - Determining resource availability
- Work Accomplishments
- *Future Work*

Future Work

- Pcall and plet don't have early termination cases. The parallelism overhead could be reduced if the early termination code was removed for these two primitives. The need to handle user-level interrupts mitigates the benefits gained from removing the early termination code.
- I kind of wish pcall was called par, or that pcall was structured like funcall. It took me awhile to come around to this viewpoint. Maybe I really want pfuncall. What exists right now is really peval, except macros are disallowed.
- Have the producer immediately acquire a piece of work from the work queue instead of waiting for the children to finish. This is messy when thinking about early termination, so it's been avoided so far.

Future Work (cont'd)

- Create a futures interface
  - There's nothing simpler than the identity function
    - Would allow parallel evaluation of only some let bindings instead of all of them without requiring two let statements
- Parallel garbage collectors

Conclusion

- Parallelism Primitives
- Evaluation Strategy
  - Producer/Consumer roles and implementation
  - Early termination implementation
  - Determining resource availability
- Work Accomplishments
- Future Work

Questions