Functional Languages and Parallelization

Instantiating Ideas with ACL2

“From this we see that the structure of the data can serialize a process” - *Gabriel and McCarthy 1984*

July 20, 2005

David L. Rager

Overview

- Constructs Added (plet, parallelize, pand, por)
- Discussion of Efficiency
  - Top level vs. recursive parallelization
  - Tests (Fib, Mergesort, Adaptive, Sum-tree)
- Loose-ends
  - Interrupting parallelization
  - Shared memory vs. distributed
  - LISPs' current thread support
- Problem of Granularity
- Future Work

Constructs Added (Plet)

- Computes the bindings in parallel
  - Ex: (plet ((x 4) (y 5)) (+ x y))
- Logically let
- Captures lexical and special variables
  - demo ptest-forms.lisp
- Stobjs
  - demo ptest-stobj.lisp
  - Safe because stobj read/writes must follow all syntax properties of a let, which is heavily restricted
- Plet* doesn’t exist for obvious reasons

Construct Added (Parallelize)

- Computes the arguments to the function in parallel
  - Ex: (parallelize (+ 3 4))
- Supposed to be “easy to use” – just “wrap it around”
- Similar in meaning to pcall in other papers
  - Since it’s different structure, it currently retains a different name
  - Pcall more intuitive name?

Construct Added (Pand/Por)

- Computes the arguments to and/or in parallel
  - Ex: (pand 3 4 5 nil 7) (por 3 nil 5 6 7)
- Is it possible to do lazy evaluation? No!
- Do we exit early upon knowing the rest of the computation is irrelevant? Yes!
- Was that tricky? Yes.
  - How do we cancel previously spawned threads? We don’t.
  - We remove closures from the closure queue, and since those closures’ results are meaningless, we signal the thread waiting for results to compute its result. When that thread goes to look up the result, it will find a nil, but since the result doesn’t matter, this is OK.
  - With aggressive parallelization, early exiting is effectively disabled
- Might be encountering an OpenMCL bug with early exit (may be in my code too)
- Easy to provide constructs without early exit
- Current bug: since and/or are macros, they can’t be applied, so we currently use eval, which breaks parallelize sometimes – NOT a big deal, just disclaimed to give an accurate snapshot of the project

Discussion of Efficiency

- Top Level Parallelization
  - Presented in May 2005
  - Parallelizes once at the top level
  - Ex: foo

```
(defun foo (x)
 (if (zp x) 0
     (plet ((a x)
            (b (foo (- x 1))))
           (+ a b))))
```

    - The first call to foo will actually result in computing the binding for A and the binding for B in parallel
    - The recursive calls (foo (- x 1)), (foo (- x 2)),… (foo 0) will treat the plet as a let
  - Good for simple problems like Mergesort on a fixed number of cores/processors

Discussion of Efficiency

- Recursive Parallelization
  - New
  - Parallelizes at all levels
  - Ex: foo

```
(defun foo (x)
 (if (zp x) 0
     (plet ((a x)
            (b (foo (- x 1))))
           (+ a b))))
```

    - The first call to foo will actually result in computing the binding for A and the binding for B in parallel
    - The recursive calls (foo (- x 1)), (foo (- x 2)),… (foo 0) will also result in parallel computation of bindings A and B
  - Good for problems where the parallelize construct is not part of the expensive operation (see ptest-plet in ptest-adaptive.lisp)
  - Very Bad for Mergesort and anything we're likely to use plet for

Problems in Parallelizing Functional Programs

- Recursive parallelization give us **parallelization for data dependent computation**
  - Ex: count

```
(defun count (x)
  (if (atom x)
      1
     (+ (count (car x))
        (count (cdr x)))))
```

  - Will counting the car or the cdr take longer? Determining this answer ahead of time is NP complete.
  - Only recursive parallelization assumes everything will take a long time and spawns threads for all of it

Discussion of Efficiency

- Recursive Parallelization, when Resources are Available
  - New
  - **If resources are available**, can reparallelize at that level
  - Ex: Foo
    - Suppose “resources are available” for the initial call (as they should be), it will treat the call for (foo x) as a plet
    - Now, suppose it takes a long time to compute binding A, and resources are no longer “available”
    - (foo (- x 1)) will just use a let, bypassing reparallelization
    - Now, suppose resources become available during computation for (foo (- x 1))
    - For the recursive call (foo (- x 2)), we will reparallelize the computation of the bindings
    - And so forth until we finish all computation

Discussion of Efficiency

- How do we know when resources are "available?"
  - For now, let's just assume resources are available whenever the number of active threads in the system is less than the number of cores/processors in the system
  - This alone performs well for tasks with a good amount of cpu time between evaluations of the parallelization constructs
  - We will see this come back to haunt us in the granularity discussion

Tests

- Simple Arithmetic
  - ptest-arithmetic.lisp

Basic Arithmetic Done in Parallel

ptest-arithmetic.lisp

	Trial 1	Trial 2	Trial 3	Avg	Speedup
Arithmetic (10)	20.010	20.010	20.030	20.017	
Arithmetic (10) Parallel	10.060	10.030	10.050	10.047	1.992

  - note to presenter: demo with (arithmetic 3)

Tests

- Mergesort
  - ptest-mergesort.lisp
  - No longer drops elements (mv-let thread-safe now) thanks Matt!

<u>Mergesort Done in Parallel</u>

ptest-mergesort.lisp

	Trial 1	Trial 2	Trial 3	Avg	Speedup
MergeSort (1000)	0.005	0.005	0.005	0.005	
MergeSort Parallel (1000)	0.050	0.030	0.032	0.037	0.134
MergeSort Parallel (1000) with granularity function	0.005	0.005	0.006	0.005	0.938
MergeSort (50000)	0.416	0.421	0.414	0.417	
MergeSort Parallel (50000)	0.630	0.536	0.694	0.620	0.673
MergeSort Parallel (50000) with granularity function	0.249	0.253	0.247	0.250	1.670

Note: Since mergesort is GC intensive, I disabled the GC for these tests. GC time is oddly the same btw parallelized and non-parallelized executions

Note2: All these tests were run in raw OpenMCL, due to stack limitations

Note3: We'd really like to test lists larger than 50000 elements, but we have stack limitations still

- Note to presenter: demo with (integers 50000 nil)

Tests

- Adaptive Parallelization
  - ptest-adaptive.lisp
  - Thanks to our “resources-available” idea, we can reparallelize when cores/processors go idle
  - Observe the difference between putting recursive call first and putting recursive call last
    - We could create a special version of defun that looked for the recursive call and reordered it, but *yuck*
  - Whenever por returns early, it currently causes an error – this is somewhat encouraging, because it means por does occasionally return early, even under heavy parallelization

  - Note to presenter: demo with (test-let2, test-plet2-PR, test-plet2-PL 5) and traces

Tests

- Adaptive Parallelization

Adaptive Computation Tests *ptest-adaptive.lisp*					
	Trial 1	Trial 2	Trial 3	Avg	Speedup
Let (10)	55.065	55.074	55.061	55.067	
Plet-PL (10)	29.394	29.651	29.551	29.532	1.865
Plet-PR (10)	29.418	29.474	29.474	29.455	1.869
Let (10) Two branches	110.119	110.126	110.125	110.123	
Plet-PL (10) Two branches	91.303	91.341	91.487	91.377	1.205
Plet-PR (10) Two branches	58.148	58.201	58.268	58.206	1.892
Let (10) Three branches	165.190	165.180	165.219	165.196	
Plet-PL (10) Three branches	136.999	136.806	136.916	136.907	1.207
Plet-PR (10) Three branches	87.517	87.627	87.479	87.541	1.887
And (10) Two branches	110.037	110.122	110.137	110.099	
Pand-PR (10) Two branches	58.502	58.976	58.786	58.755	1.874
Or (10) Two branches	10.015	10.013	10.013	10.014	
Por-PR (10) Two branches	58.360	58.433	bomb	58.397	0.171

Tests

- Fibonacci
  - ptest-fib.lisp
  - Thanks to our “resources-available” idea, we can reparallelize when cores/processors go idle
  - Much better results with “granularity function”

  - Note to presenter: demo (fib 34) with the granularity function and without the granularity function

Tests

- Fibonacci

Parallel Fibonacci
ptest-fib.lisp

	Trial 1	Trial 2	Trial 3		
Fib (36)	1.644	1.642	1.642	1.643	
Pfib (36)	16.719	32.047	crash	24.383	0.067
Pfib (36) with granularity function	1.239	1.213	1.171	1.208	1.360
Fib (40)	11.286	11.281	11.284	11.284	
Pfib (40)	Blows up	Blows up	Blows up		
Pfib (40) with granularity function	7.072	7.089	7.09	7.084	1.593
Fib (45)	123.968	123.976	123.974	123.973	
Pfib (45)	Blows up	Blows up	Blows up		
Pfib (45) with granularity function	76.593	75.575	75.862	76.010	1.631
Fib (47)	324.653	324.645	324.66	324.653	
Pfib (47)	Blows up	Blows up	Blows up		
Pfib (47) with granularity function	198.912	198.52	198.569	198.667	1.634

- Note to presenter: demo (fib 34) with the granularity function and without the granularity function

Tests

- Sum a tree

```
(defun sum (tree)
 (cond ((null tree) 0)
       ((atom tree) tree)
       (t (+ (sum-tree (car tree))
             (sum-tree (cdr tree)))))
```

  - Parallel version absolutely bombs
  - There’s no quick way to estimate granularity for a function that’s so fast
  - It’s spawning a huge number of processes for some reason right now – not sure why
  - Handling this function may be a future goal, but since it’s so fast, it would need to be rewritten before we could save any time

Loose-Ends

- Interrupting Parallelization
  - Most parallelization variables are reset upon reentering of the ACL2 loop
  - Since we spawn a thread to process closures and spawn other threads, we don’t reset the variables related to that
  - Killing that thread would be ulgy (although I have done it automatically in the past)

Loose-Ends

- Shared Memory vs. Distributed Memory
  - The test for whether resources are available requires quick access to shared data
  - While a lock is not required, since it’s just an estimate, the question of whether resources are available is asked a lot
  - Results: fine for SMP, most likely horrendous for distributed memory
  - Conclusion: default to top-level parallelization in distributed memory architectures
    - Maybe could give each spawned process a bank of resources to work with and send updates to that number asynchronously

Problem of Granularity

- Large granularity – something that takes a long time
- Small granularity – something that takes a short time
- So why do we even need the “granularity function” for Fibonacci?
  - Resources become available to other recursive calls at random times
  - Therefore we can be anywhere in a recursive call stack
  - We’re most likely to be at the points where we’re taking fib of smaller numbers, because there are more calls to them
  - As a result the smaller calls have a built-in advantage to asking if resources are available and then seizing those resources
  - Spawning threads for small function calls is a waste of time
  - OpenMCL can’t handle threads spawning and dying so quickly and creation/destruction of semaphores so quickly
  - Without it, even though only eight may be active at any moment, we create thousands of threads across a function call

Problems with "Granularity Function"

- The user would prefer to not enter the "granularity function"
- Evaluating the "granularity function" takes time
- Getting rid of it and having perfect results is NP Hard (I think)
- How can we approximate the granularity function automatically?

Approximating Large Granularity

- Concept 1: Give each recursive call a rank
  - Begin with rank 0
  - Rank would be an optional argument to plet, etc.
  - Each recursive call gets (+1 current-rank)
  - The lower the rank the larger your granularity
  - Use *randomization* to give each process a chance to reparallelize based upon its rank
  - Let *x* be the # of expected reparallelizations
    - Chance of reparallelization for a process = (1 / rank) * x
    - Might be off by a factor of 2, but this is the general idea
  - This will work terribly when we recur heavily on the cdr of a list

Approximating Large Granularity

- Using Rank
  - If I am the maximum rank of the last 20 parallelization requests, I get to reparallelize
    - Works well for fib,
    - Never get to reparallelize with mergesort – might be okay
  - If I am the max…, *or* there hasn't been reparallelization for the last 50 calls, and I am "near the top" in rank, I get to reparallelize

Approximating Large Granularity

- ACL2 actually has a builtin granularity function. What’s it called? *The measure*.
- Keep track of the measure during execution to get an idea of what a “big” measure is
- Derive a threshold based upon a weighted average, where the weights are determined by being above or below the regular average

Approximating Large Granularity

- What ideas do you all have?

Where Next?

- Make running closures even cheaper
  - Keep lists of spare processes and semaphores
  - Pull from this queue instead of destroying old processes and semaphores and creating new ones
  - To a degree, having a semi-heavy mechanism for creating threads is good, because it keeps me honest

Where Next?

- Fix pand/por bugs
- Conduct experiments on optimal threads allowed versus the number of parallelization closures allowed to enqueue
- Evaluate on 4-way machines
- Continue researching previous work
- Port concept to recursive functions in Java – would open a whole new world
- Tracing
- Summing a tree

Where Next?

- Integrate parallelization constructs into the theorem prover itself
  - Parallelization constructs must most likely go through a “proving” time before being integrated so strongly
    - Who wants a rollback? Definitely not Dr. Moore and Dr. Kaufmann.
  - Rewriter
  - Waterfall – requires doing something with output
  - Rule matching/free variable matching
  - Relieving hypothesizes

Overview

- Constructs Added (plet, parallelize, pand, por)
- Discussion of Efficiency
  - Top level vs. recursive parallelization
  - Tests (Fib, Mergesort, Adaptive, Sum-tree)
- Loose-ends
  - Interrupting parallelization
  - Shared memory vs. distributed
  - LISPs’ current thread support
- Problem of Granularity
- Future Work