Implementing a Parallelism Library for a Functional Subset of LISP

The International LISP Conference 2009
Cambridge, Massachusetts

David L. Rager (ragerdl@cs.utexas.edu)
Warren A. Hunt Jr. (hunt@cs.utexas.edu)

Problem Statement

- How do we take advantage of multi-core processors in LISP computation today?
- Our approach:
  - Pick a functional and useful subset of LISP, the ACL2 theorem prover
    - ACL2 users simulate industrial models, so performance is important
    - Eventual opportunity to parallelize the theorem prover itself
  - Through our four semantically side-effect-free parallelism primitives, multiple cores can be kept busy
    - By having semantically side-effect-free parallelism primitives, the user is unburdened with reasoning about parallel evaluation mechanisms such as threads and synchronization variables

Overview

- Related work
- Four primitives: *plet*, *pargs*, *pand*, *por*
- Three performance features
  - Data dependent parallelism
  - Granularity forms
  - Early termination
- Overview of Implementation
- Performance Results

Related Work

- Futures
  - (future α) added the computation α to a work queue evaluated in parallel
  - Insufficient to implement our library because:
    - No notion of stopping parallel evaluations once they're started
    - Implementing futures requires modifying the LISP

Primitives: *plet*

- Semantically: let
- Implementation: (1) evaluate the bindings in parallel (2) apply a closure containing the body, with the results from the parallel evaluation as arguments to the closure
- Simple example:

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

Primitives: *pargs*

- Semantically: the identity macro
- Implementation: (1) evaluate the arguments to the function surrounded by pargs in parallel (2) apply the function to the results of those evaluations
- Simple example:

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

Primitives: *pand* / *por*

- Semantically: Boolean-valued and/or that evaluate eagerly
  - E.g., in *(pand x y)*, the resulting value of *x* does not guard the evaluation of *y*.
- Implementation: (1) evaluate *all* of the arguments to the pand/por in parallel (2) apply a function that returns the Boolean and/or of its arguments
- Simple example:

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

User-level Features: *data-dependent parallelism*

- The problem: Can we allocate our CPU resources statically? No.

- Example:

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

  - Should we use pcount in the recursive call for car, cdr, or both?

User-level Features: *data-dependent parallelism*

- Our solution: Both. Define functions recursively and dynamically evaluate CPU resource availability.

- Example:

```
(defun pcount (x)
  (if (atom x) 1
     (pargs (+ (pcount (car x))
               (pcount (cdr x))))))
```

  - With every recursive call, pargs parallelizes iff there are resources available

User-level Features: *granularity form*

- The problem: How can we avoid parallelism overhead for data that is "small" - work that is small in granularity

- Example:

```
(defun pfib (x)
  (cond ((<= x 0) 0)
        ((= x 1) 1)
        (t (pargs (+ (pfib (- x 1))
                     (pfib (- x 2)))))))
```

  - E.g., we don't want to parallelize when x is less than 30.

User-level Features: *granularity form*

- Our solution: Give the programmer a *granularity form* to specify a “minimum size” for the data before parallelizing computation
- When the evaluation of the granularity form returns nil, the parallelization primitive converts to its serial equivalent
- Example:

```
(defun pfib (x)
  (cond ((<= x 0) 0)
        ((= x 1) 1)
        (t (pargs (declare (granularity-form (> x 30)))
                  (+ (pfib (- x 1))
                     (pfib (- x 2)))))))
```

User-level Features: *early termination*

- The problem: We expect our library to either beat or at worst, match the performance of if's lazy evaluation.

- Example:

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

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

  - If the recursive car call returns nil and we have already begun evaluating the cdr recursion, we need a way to abort it.

User-level Features: *early termination*

- Our solution: When computing the conjunction of objects by using pand, once one of those objects evaluates to nil, all sibling computations are aborted and the pand returns nil

- Can gain super linear speedup when a nil is beyond the first argument to the pand
  - E.g. (pand α β γ δ), where γ quickly evals to nil

- Similar application to por, except the early termination condition is when an argument evaluates to non-nil and the result returned is t

Implementation

- LISP threading interface
- Work producers (pargs, plet, pand, por)
- Work consumers (threads)

Note: Only implemented on CCL and SBCL

LISP Threading Interface

- Mostly direct translation from our interface to LISP-level primitives
  - Thread handling – ability to create, kill, and interrupt threads
  - Mutual exclusion and signaling – locks, semaphores, condition variables
    - CCL implementation extended by Gary Byers with a *semaphore notification object*

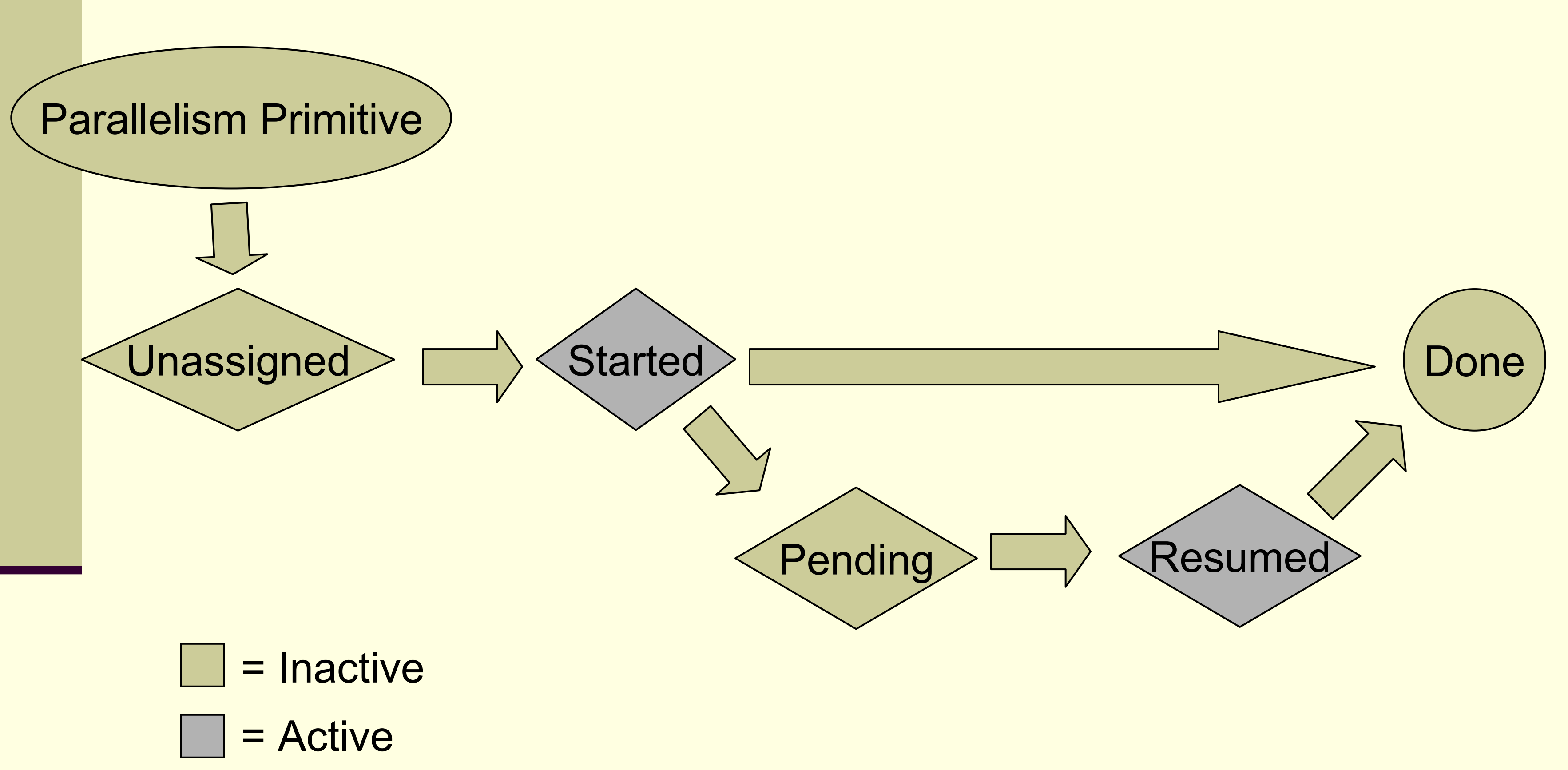
Parallelism Producers

- Primitives are the producers
- Parallelization of a primitive requires three conditions:
  - Available CPU and thread resources
  - The arguments to the primitive must exceed a certain “size”
  - Parallelism must be enabled

Producing Work

- If there are resources available and the work is deemed to be of large enough granularity, the parallelism primitive gathers data necessary to parallelize computation, e.g.:
  - Closures for each computation
  - An array for keeping track of parent-child thread relationships
  - An array for sharing the results of closure evaluations
- These data are stored in pieces of parallelism work and placed on the global *work queue*
- After work is placed on the *work queue*, parallelism consumers are created if necessary, signaled, and begin evaluating the work

Life Cycle of a Piece of Parallelism Work



Ideal Amounts of Work by Classification (intro slide)

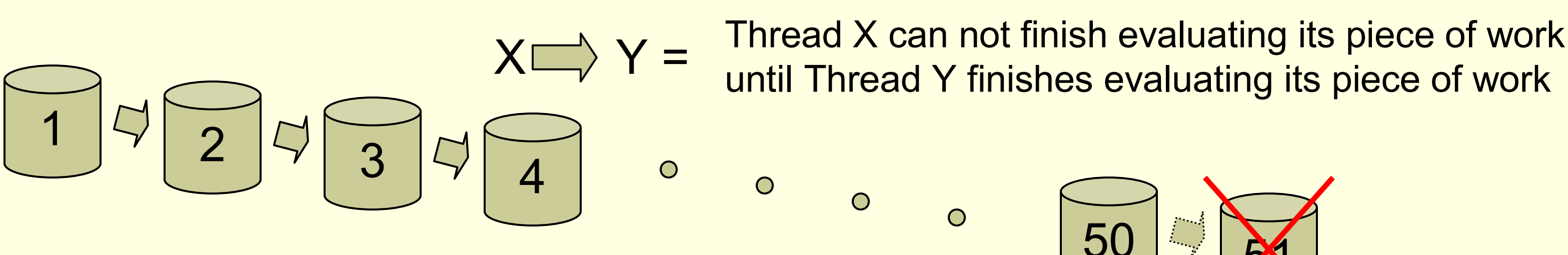
- Let p be the number of CPU cores in the system
- Let l be the maximum number of threads reliably supported by the underlying LISP
- We strive to maintain the following invariant:

Unassigned	Started + Resumed	Pending
— $\simeq p$ —	— $p \leq$ && $\leq 2p$ —	-irrelevant -

| — $\leq l$ — |

Estimating Resource Availability

- The total number of pieces of parallelism work in the system must be limited to the number the underlying LISP can support
  - Note: every piece of parallelism work *must* eventually be evaluated by a work consumer. If 50 pieces of parallelism work mostly depend on each other, and spawning 51 threads causes the LISP to crash, a request to add a piece of parallelism work that requires a 51st thread must be denied



Consuming Work

- Labeling CPU core and thread states
  - CPU cores are either idle or active
  - Threads are in one of three states:
    - Idle – waiting for a CPU core to be free or for parallelism work to enter the *work queue*
    - Active – allocated a CPU core and actively evaluating a piece of parallelism work
    - Pending – encountered a parallelism primitive and decided to parallelize evaluation of that primitive

Matching Up States

Work State	Unassigned	Started	Pending*	Resumed*
Thread State	n/a	active	pending	active
Allocated Core	n/a	yes	no	yes

*the pending and resumed states are not always entered

Granting a Thread the Green Light (Part I)

- Two invariants:
  - To keep CPU cores busy, the number of active threads should be at least the number of CPU cores (assuming there is enough parallelism work available)
  - To limit thread thrashing, the number of active threads should never be more than twice the number of CPU cores in the system (this limit is somewhat arbitrary)

Granting a Thread the Green Light (Part II)

- It is good to give resuming pieces of work higher priority access to CPU core resources than just starting threads
  - Threads become available for other pieces of parallelism work sooner
  - Often application of a parallelism primitive's closure or function takes relatively little time

Granting a Thread the Green Light (Part III)

- How we progress and still maintain the previously mentioned invariants:
  - A thread will begin evaluating a fresh piece of parallelism work whenever the number of active worker threads < the number of CPU cores
  - A thread will resume evaluation of a previously-pending piece of parallelism work whenever the number of active worker threads < 2x the number of CPU cores
    - In the paper, this second criterion is named a “resumptive heuristic”

Ideal Amounts of Work by Classification (summary slide)

- Let p be the number of CPU cores in the system
- Let l be the maximum number of threads reliably supported by the underlying LISP
- We strive to maintain the following invariant:

Unassigned	Started + Resumed	Pending
— $\simeq p$ —	— $p \leq$ && $\leq 2p$ —	-irrelevant -
————	$\leq l$	————

Optimizations

- Semaphore recycling
  - Unclear whether still necessary but definitely important at some point of the project's life
- Thread recycling
  - Takes about four times longer to evaluate constant arguments to "+" in parallel without thread recycling
- Granularity test costs
  - Since granularity tests need to be fast regardless, we currently evaluate the granularity form before testing for resource availability

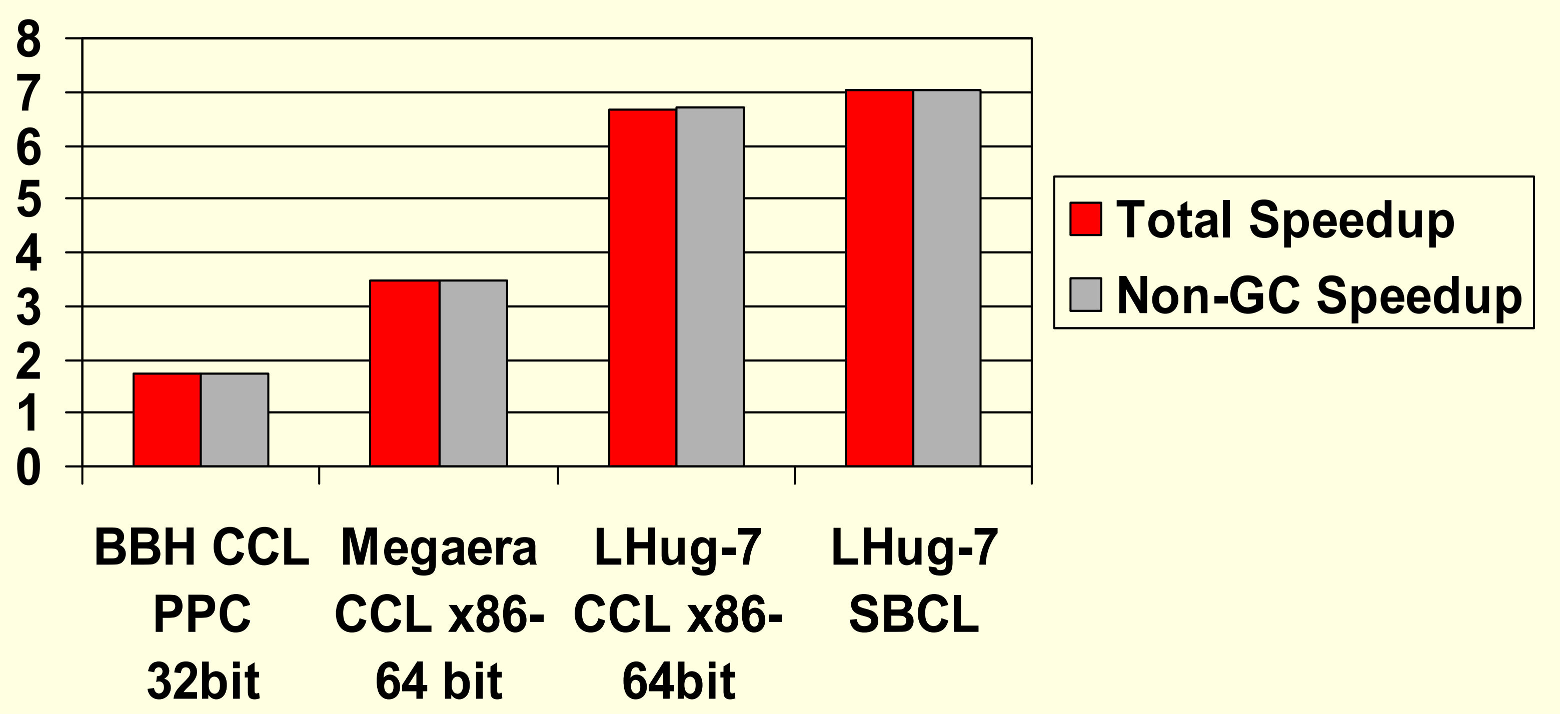
Performance Results

- Machines
  - BBH (2 cores): 64 bit PPC; CCL 32 bit PPC
  - Megaera (4 cores): 64 bit x86; CCL 64 bit x86
  - LHug-7 (8 cores): 64 bit x86; CCL 64 bit x86 & SBCL 64 bit, threaded build

Performance Results

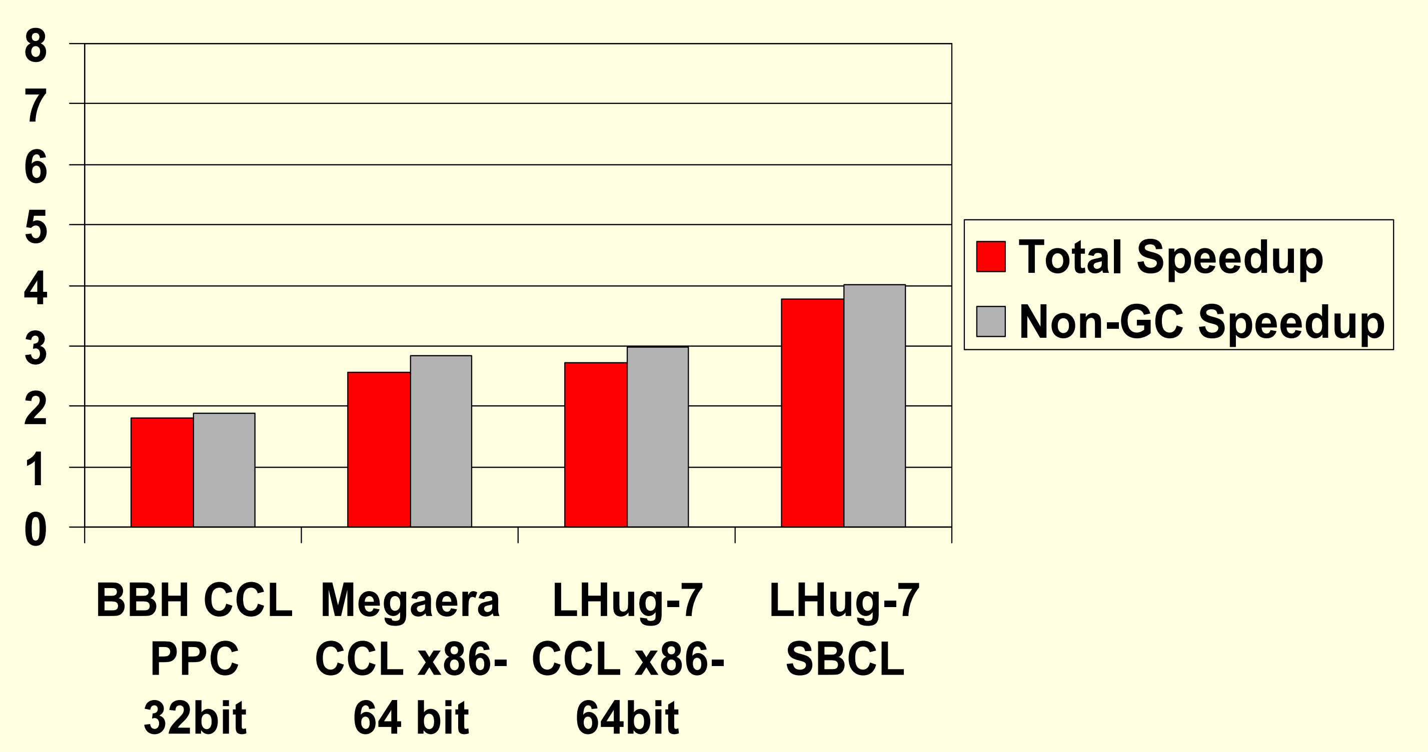
- Tests
  - Doubly recursive Fibonacci function
  - Matrix multiplication
  - Mergesort

Performance Results for Fibonacci



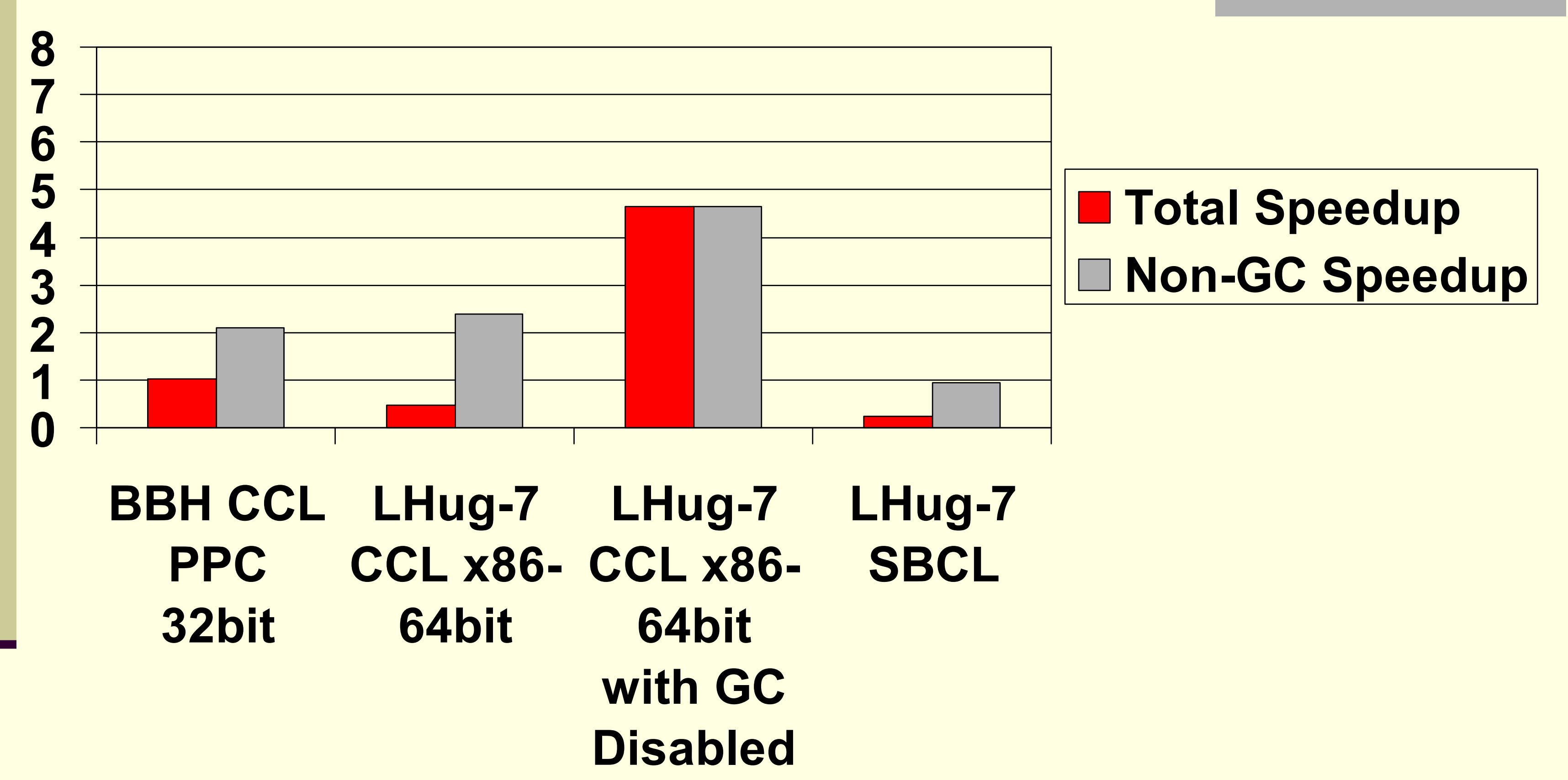
- Determines speedup for the most basic parallelizable functions
- Maximum speedup of 7x
  - Non-8x speedup probably partly due to the need to test the granularity form. Fibonacci itself is quite fast!

Performance Results for Matrix Multiplication



- Both algorithms take advantage of cache effects, as documented by Bryant
- Loss of half the potential speedup is likely caused by the difference in the algorithms
  - Parallel version must split the problem hierarchically

Performance Results for Mergesort



- Generates immense amounts of garbage
- In the best case, garbage collection limits potential speedup of about 4x

Future Work

- Port to other LISPs
- Update our multi-threading interface to take advantage of more recent LISP features. E.g., semaphores in SBCL
- Use library in the ACL2 mechanized prover's implementation

This work would benefit from parallel garbage collection.

Conclusion

- Parallelism primitives: *plet*, *pargs*, *pand*, *por*
- User-level features
  - Data-dependent parallelism
  - Granularity forms
  - Early termination
- Implementation
  - LISP multi-threading interface
  - Producing work
  - Consuming work
- Performance Results

Questions

Appendix Slides

Labeling the States of Pieces of Parallelism Work

- Evaluations to parallelize are broken down into pieces of parallelism work
- A piece of parallelism work can be in one of four stages:
  - Unassigned – not yet acquired by a consumer
  - Started – acquired by a consumer and being processed
  - Pending – acquired by a consumer but waiting for children to finish evaluating
  - Resumed – acquired by a consumer, children have finished evaluating, and finishing evaluation of itself
- The started and resumed pieces are considered active, while the unassigned and pending pieces are considered inactive

Estimating Resource Availability (# of unassigned pieces ~=p)

- The number of pieces of parallelism work available to be processed should be approximately equal to the number of CPU cores in the system
- Ensures that if all consumers finish evaluating their piece of parallelism work near simultaneously, that the CPU core resources will still be used

Performance Results (Fibonacci)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
BBH CCL PPC 32bit					
Serial	279.05	0.00	279.05		
Parallel	161.23	0.02	161.21	1.73	1.73
Megaera CCL x86-64 bit					
Serial	160.87	0.00	160.87		
Parallel	46.15	0.01	46.14	3.49	3.49
LHug-7 CCL x86-64bit					
Serial	192.31	0.00	192.31		
Parallel	28.78	0.09	28.69	6.68	6.70
LHug-7 SBCL					
Serial	462.89	0.00	492.07		
Parallel	65.76	0.01	65.75	7.04	7.04

- Maximum speedup of 7x
  - Non-8x speedup probably due to the need to test the granularity form. Fibonacci itself is quite fast!

Performance Results (Matrix Multiplication)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
BBH CCL PPC 32bit					
Serial	154.72	7.90	146.83		
Parallel	85.73	7.75	77.98	1.80	1.88
Megaera CCL x86-64 bit					
Serial	82.99	4.21	78.78		
Parallel	32.30	4.51	27.79	2.57	2.83
LHug-7 CCL x86-64bit					
Serial	145.08	6.58	138.50		
Parallel	53.03	6.83	46.46	2.72	2.98
LHug-7 SBCL					
Serial	172.43	3.31	169.13		
Parallel	45.73	3.52	42.21	3.77	4.01

- Both algorithms take advantage of cache effects, as documented by Bryant
- Loss of half the potential speedup is likely caused by the difference in the algorithms
  - Serial version can just cdr down lists
  - Parallel version must split the problem hierarchically

Performance Results (Mergesort)

Case	Total Time	GC Time	Non-GC Time	Total Speedup	Non-GC Speedup
BBH CCL PPC 32bit					
Serial	26.13	13.92	12.21		
Parallel	25.46	19.69	5.80	1.03	2.11
LHug-7 CCL x86-64bit					
Serial	181.50	155.49	26.01		
Parallel	390.63	379.73	10.90	0.46	2.39
LHug-7 CCL x86-64bit with GC Disabled					
Serial	30.81	0.00	30.81		
Parallel	6.62	0.00	6.62	4.65	4.65
LHug-7 SBCL					
Serial	124.77	92.47	32.30		
Parallel	529.85	495.45	34.40	0.24	0.94

- Generates immense amounts of garbage
- In the best case, garbage collection limits potential speedup of about 4x