Manticore:
A heterogeneous parallel language

Matthew Fluet
Rochester Institute of Technology

March 30, 2012
Heterogeneous Parallelism

Manticore is a *heterogeneous parallel* programming language aimed at general-purpose applications running on multi-core processors.

- hardware supports parallelism at multiple levels
- software exhibits parallelism at multiple levels
- to maximize productivity and performance, programming languages must support parallelism at multiple levels
- We call this property *heterogeneous parallelism*

http://manticore.cs.uchicago.edu
People and Acknowledgements

The Manticore Project is a joint project between researchers at the University of Chicago and the Rochester Institute of Technology:

- Sven Auhagen — University of Chicago
- Lars Bergstrom — University of Chicago
- Matthew Fluet — Rochester Institute of Technology
- Mike Rainey — Max Planck Institute for Software Systems
- John Reppy — University of Chicago
- Adam Shaw — University of Chicago
- Yingqi Xiao — University of Chicago

and supported (in part) by the

- National Science Foundation
Parallelism in Hardware

Hardware supports parallelism at multiple levels:

- single instruction, multiple data (SIMD) instructions
- simultaneous multithreading executions
- multicore processors
- multiprocessor systems
Parallelism in Software

Software exhibits parallelism at multiple levels.
Consider a networked flight simulator:

<table>
<thead>
<tr>
<th>Flight Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User Interface</strong></td>
</tr>
<tr>
<td>sound</td>
</tr>
<tr>
<td>keyboard</td>
</tr>
<tr>
<td>mouse</td>
</tr>
<tr>
<td><strong>Physics Simulation</strong></td>
</tr>
<tr>
<td>Particle Systems</td>
</tr>
<tr>
<td><strong>Artificial Intelligence</strong></td>
</tr>
<tr>
<td>Artificial Intelligence</td>
</tr>
<tr>
<td><strong>Network</strong></td>
</tr>
<tr>
<td>server</td>
</tr>
<tr>
<td>player 2</td>
</tr>
<tr>
<td>player 3</td>
</tr>
<tr>
<td><strong>Graphics</strong></td>
</tr>
</tbody>
</table>
Parallelism in Software

Software exhibits parallelism at multiple levels.

Consider a networked flight simulator:

- **User Interface**
  - sound
  - keyboard
  - mouse

- **Network**
  - server
  - player 2
  - player 3

- **Physics Simulation**
  - Particle Systems
    - rain, fog, clouds

- **Artificial Intelligence**

- **Graphics**

- SIMD parallelism for physics simulation
Parallelism in Software

Software exhibits parallelism at multiple levels.

Consider a networked flight simulator:

- data-parallel computations for particle systems to model natural phenomena (e.g., rain, fog, and clouds)
Parallelism in Software

Software exhibits parallelism at multiple levels.
Consider a networked flight simulator:

- Parallel threads for preloading terrain and computing level-of-detail refinements
Parallelism in Software

Software exhibits parallelism at multiple levels.
Consider a networked flight simulator:

speculative search for artificial intelligence
Parallelism in Software

Software exhibits parallelism at multiple levels.

Consider a networked flight simulator:

- parallel threads for user interface and network components
Manticore: A Heterogeneous Parallel Language

- An effort to design and implement a parallel functional programming language supporting **heterogeneous parallelism**:
  - commodity applications with multiple levels of software parallelism
  - commodity hardware with multiple levels of hardware parallelism
  - maximize productivity and performance
  - balance programmer and compiler effort
Manticore: A Heterogeneous Parallel Language

A long-range project with two major aspects:

- Language design for heterogeneous parallel programming.
  - DAMP’07, ML’07, ICFP’08a, JFP’11

- Language implementation for heterogeneous parallelism.
  - POPL’07, DAMP’08, ICFP’08b, ICFP’09, ICFP’10, MSPC’11, JFP’12
Manticore: Language Design

Combination of three distinct, but synergistic, sub-languages:

- A mutation-free subset of Standard ML for *sequential* programming
  - functional programming

- Language mechanisms for *explicitly-threaded* parallelism
  - programmer explicitly spawns threads
  - coordinate via synchronous message-passing

- Language mechanisms for *implicitly-threaded* parallelism
  - programmer annotates fine-grained parallel computations
  - compiler and runtime map onto parallel threads
Sequential Programming

Rooted in the family of statically-typed, strict functional languages, such as OCaml and Standard ML

- Functional languages emphasize a value-oriented and mutation-free programming model
  - avoids entanglements between separate computations

- Strict languages (rather than lazy or lenient languages) are easier to implement efficiently and are accessible to a larger community of potential users
Explicitly-threaded Parallelism

Language mechanisms for *explicitly-threaded* parallelism

- programmer explicitly spawns threads
- coordinate via synchronous message-passing

These explicit mechanisms serve two purposes:

- support concurrent programming
  - an important feature for systems programming
- support explicit-parallel programming
  - for additional programmer control
Explicitly-threaded Parallelism

The explicitly-threaded parallelism mechanisms of Manticore are based on those of Concurrent ML (CML).

- dynamic creation of threads and typed channels
- rendezvous communication via synchronous message passing
- first-class synchronous operations, called events
  - support building synchronization and communication abstractions
- automatic reclamation of threads and channels
- pre-emptive scheduling of explicitly concurrent threads
- efficient implementation
  - both uni- and multi-processors (see POPL’07, DAMP’08, & ICFP’09)
Explicitly-threaded Parallelism

- Explicit thread creation
  
  ```
  spawn exp
  ```

- Channels for synchronous-message passing
  
  ```
  type 'a chan
  val channel : unit -> 'a chan
  val send : ('a chan * 'a) -> unit
  val recv : 'a chan -> 'a
  ```

- The CML-style event combinators
  
  ```
  type 'a event
  val sync : 'a event -> 'a
  val sendEvt : ('a chan * 'a) -> unit event
  val recvEvt : 'a chan -> 'a event
  val choose : 'a event * 'a event -> 'a event
  val wrap : 'a event * ('a -> 'b) -> 'b event
  val guard : (unit -> 'a event) -> 'a event
  ```
Implicitly-threaded Parallelism

Language mechanisms for *implicitly-threaded* parallelism

- programmer annotates fine-grained parallel computations
- compiler and runtime map onto parallel threads

Implicitly-threaded parallelism is more specific (and less expressive) than explicitly-threaded parallelism, but

- express common idioms of parallel computation
- ease the burden for both programmer and compiler
  - programmer able to utilize simple parallel constructs: efficiently (in terms of program text) express the desired parallelism
  - compiler able to analyze and optimize simple parallel constructs: efficiently (in terms of time and computational resources) execute
Implicitly-threaded Parallelism

Manticore provides several light-weight syntactic forms for introducing implicitly-parallel computations.

These forms are *hints* to the compiler and runtime that a computation is a good candidate for parallel execution.

- *Parallel seqs*: fine-grain data-parallel computations over sequences
- *Parallel tuples*: basic fork-join parallel computation
- *Parallel bindings*: data-flow and work-stealing parallelism
- *Parallel case*: non-deterministic speculative parallelism
- *Cancellation*: unused/abandoned subcomputations

(see ICFP’08a, JFP’11)
Parallel Seqs

Support for parallel computations on arrays and matrices is common in parallel languages.

Operations on arrays and matrices naturally express data parallelism.

- a single computation that is performed in parallel across a large number of data elements

Manticore adopts the nested parallel array mechanism of NESL

type 'a parray

- immutable sequences that can be computed in parallel
- nested data parallelism
  - arbitrary element types: arrays of floating-point numbers, arrays of user-defined datatypes, arrays of arrays
Parallel Seqs

Parallel sequence comprehension form (NESL/Nepal/DPH):

```
[| exp | pat_i in exp_i; where pred |]
```

For example, the parallel point-wise summing of two seqs:

```
[| x + y | x in xs, y in ys |]
```

**NOTE:** zip semantics, not Cartesian-product semantics.

**NOTE:** no mutation, no constant-time random-access
Parallel Seqs (continued ...)

Mandelbrot set computation:

```ml
fun x i = x0 + dx * itof i
fun y j = y0 - dy * itof j
fun loop (cnt, re, im) = 
  if (cnt < 255) andalso (re*re + im*im > 4.0) 
    then loop(cnt+1, re*re - re*im + re, 2.0*re*im + im) 
    else cnt

val L = [| 0..N |]
val image = 
  [| [| loop(0, x i, y j) | i in L |] | j in L |]
```
Parallel Tuples

Parallel tuples provide fork-join parallelism. Consider adding the leaves of a binary tree.

...
Parallel Tuples

Parallel tuples provide fork-join parallelism. Consider adding the leaves of a binary tree.

\[
\text{datatype tree} = \text{Lf of int} | \text{ND of tree * tree}
\]

\[
\text{fun treeAdd (Lf n) = n}
| \text{treeAdd (ND(t1, t2)) =}
\quad (\text{op +}) (| \text{treeAdd t1, treeAdd t2} |)
\]
Parallel Bindings

Parallel bindings provide data-flow parallelism. Consider multiplying the leaves of a binary tree.

```ml
fun treeMul (LF n) = n
  | treeMul (ND(t1, t2)) = let
    val b = treeMul t2
    val a = treeMul t1
    in if (a = 0) then 0 else a*b end
```
Parallel Bindings

Parallel bindings provide data-flow parallelism. Consider multiplying the leaves of a binary tree.

\[
\text{fun treeMul (LF \: n) = } n \\
\mid \text{treeMul (ND(t1, t2)) = let} \\
\quad \text{pval b = treeMul t2} \\
\quad \text{val a = treeMul t1} \\
\quad \text{in if (a = 0) then 0 else a*b end}
\]
Parallel Bindings (continued ...)

Parallel bindings provide work-stealing parallelism. Again, consider adding the leaves of a binary tree.

```
fun treeAdd (LF n) = n
  | treeAdd (ND(t1, t2)) = let
dval b = treeAdd t2
  in treeAdd t1 + b end
```

treeAdd t2 is evaluated by the current thread, while treeAdd t1 + • is placed on a work-stealing queue.
Parallel Bindings (continued …)

Parallel bindings provide data-flow parallelism.

▶ let \texttt{pval} \( x = \texttt{exp}_1 \) in \texttt{exp}_2 end
▶ mnemonic: main thread executes \( \downarrow \texttt{exp}_2 \)

Parallel bindings provide work-stealing parallelism.

▶ let \texttt{dval} \( x = \texttt{exp}_1 \) in \texttt{exp}_2 end
▶ mnemonic: main thread executes \( \uparrow \texttt{exp}_1 \)
Parallel Cases

Parallel cases support speculative parallelism, useful when we want the quickest answer (e.g., search problems).
Consider picking an arbitrary leaf of the tree:

```ml
fun treePick (LF n) = n
| treePick (ND(t1, t2)) =
  pcase treePick t1 & treePick t2
  of n & ? => n
  | ? & n => n
```
Parallel Cases

Parallel cases support speculative parallelism, useful when we want the quickest answer (e.g., search problems). Consider picking an arbitrary leaf of the tree:

```haskell
fun treePick (LF n) = n
  | treePick (ND(t1, t2)) = pcase treePick t1 & treePick t2
    of n & ? => n
    | ? & n => n
```

Can be used to define a number of common idioms:

\[
\begin{align*}
\text{e1} \ |?| \ e2 & \equiv \text{pcase e1} \ & \text{e2} \\
& \quad \text{of n} \ & ? \ => \ n \\
& \quad | \ ? \ & n \ => \ n
\end{align*}
\]

\[
\begin{align*}
\text{e1} \ |\text{andalso}| \ e2 & \equiv \ldots \\
\text{e1} \ |\text{orelse}| \ e2 & \equiv \ldots
\end{align*}
\]

\[
\begin{align*}
\text{e1} \ |\*| \ e2 & \equiv \text{pcase e1} \ & \text{e2} \\
& \quad \text{of 0} \ & ? \ => \ 0 \\
& \quad | \ ? \ & 0 \ => \ 0 \\
& \quad | \ a \ & b \ => \ a \ * \ b
\end{align*}
\]
Parallel Cases (continued ...)

Can be used to define a number of common idioms:

\[
\begin{align*}
e_{1} \mid \_\_ \mid e_{2} & \equiv \text{pcase } e_{1} \& e_{2} \\
& \quad \text{of } n \& \_\_ \Rightarrow n \\
& \quad \mid \_\_ \& n \Rightarrow n
\end{align*}
\]

\[
\begin{align*}
e_{1} \mid \text{andalso} \mid e_{2} & \equiv \cdots
\end{align*}
\]

\[
\begin{align*}
e_{1} \mid \text{orelse} \mid e_{2} & \equiv \cdots
\end{align*}
\]

\[
\begin{align*}
e_{1} \mid \_\_ \mid e_{2} & \equiv \text{pcase } e_{1} \& e_{2} \\
& \quad \text{of } 0 \& \_\_ \Rightarrow 0 \\
& \quad \mid \_\_ \& 0 \Rightarrow 0 \\
& \quad \mid a \& b \Rightarrow a \ast b
\end{align*}
\]

\[
\begin{align*}
\text{fun } \text{treeMul} \ (\text{LF } n) & = n \\
\mid \text{treeMul} \ (\text{ND}(t_{1}, t_{2})) & = \\
& \text{treeMul} \ t_{1} \mid \_\_ \mid \text{treeMul} \ t_{2}
\end{align*}
\]
Exceptions

Exceptions follow a *sequential semantics*.

```haskell
fun f n = if n = 100 then raise Foo else ...
fun g n = if n = 100 then raise Goo else ...

(| f 100, g 100 |)
handle Foo => ... (* handled Foo *)
    | Goo => ... (* unreachable *)
```
Exceptions

Exceptions follow a *sequential semantics*.

```plaintext
fun f n = if n = 100 then raise Foo else ...
fun g n = if n = 100 then raise Goo else ...

(| f 100 , g 100 |)
handle Foo => ... (* handled Foo *)
| Goo => ... (* unreachable *)
```

**NOTE:** abandoned computations are subject to *cancelation*. 
Exceptions (continued ...)

Exceptions follow a *sequential semantics*.

**NOTE:** abandoned computations are subject to cancelation.

```ml
fun f n = ... raise Foo ...
fun g n = ... raise Goo ...
fun h n = ... raise Hoo ...
```
Exceptions (continued ...)

Exceptions follow a *sequential semantics*.  
**NOTE:** abandoned computations are subject to cancelation.

```ml
fun f n = ... raise Foo ...
fun g n = ... raise Goo ...
fun h n = ... raise Hoo ...
```

Multiple forms of parallelism with cross-cutting concerns motivates the need for a common, but flexible, runtime scheduling framework.
Nested Schedulers for Heterogeneous Parallelism

Decide what work to do and when and where to do it.

Provide an *infrastructure* to support *nested* schedulers:

- core mechanisms for building schedulers
- express a variety of scheduling policies
- multiple scheduling policies in one application
- hierarchies of parallel computations

Distinguish *computation* from *computational resource*:

- Fiber – corresponds to active or suspended computation, represented as heap-allocated first-class continuation
- VProc – corresponds to a computational resource, executing at most one fiber at a time
Nested Schedulers for Heterogeneous Parallelism

Scheduling framework has proven quite flexible:

- simple round-robin thread scheduler
- engines, nested engines, workcrews/gangs, work-stealing, lazy-task creation
- (mostly) modular cancellation scheduler

(see ICFP’08b)
Nested Schedulers for Heterogeneous Parallelism

Scheduling framework has proven quite flexible:

- simple round-robin thread scheduler
- engines, nested engines, workcrews/gangs, work-stealing, lazy-task creation
- (mostly) modular cancellation scheduler

(see ICFP’08b)

Interesting directions for future research:

- domain specific language to check/enforce nested scheduler invariants; want to exclude rogue schedulers.
- implicit or explicit coupling of nested schedulers
- expressing affinity, locality, etc.
Runtime System

- Virtual Processors (VProcs)
  - hosted by a pthread and assigned to a physical core

- Garbage Collection
  - Combination of the Appel Semi-generational collector and the Doligez-Leroy-Gonthier parallel collector.
  - Each VProc has a local heap that can be independently collected.
Heap Organization

Goal: avoid synchronization & communication between VProcs.

Invariant: no pointers from global heap to local heaps
Invariant: no pointers from one local heap to another
Heap Organization

Goal: avoid synchronization & communication between VProcs.

Invariant: no pointers from global heap to local heaps
Invariant: no pointers from one local heap to another

NOTE: objects in local heap must be promoted to global heap when sent to remote thread or when accessing fiber is migrated
Lazy Tree Splitting
(see ICFP’10 and JFP’12)
Scalable and Robust Performance

Consider the problem of computing primality flags:

\[
\text{fun isPrime } i = \ldots
\]

\[
\text{val L = } [\mid 2..N \mid]
\]

\[
\text{val primalityFlags = } [\mid \text{isPrime } i \mid i \text{ in } L \mid]
\]
Scalable and Robust Performance

Consider the problem of computing primality flags:

```haskell
fun isPrime i = ...

val L = [| 2..N |]

val primalityFlags = [| isPrime i | i in L |]
```

- clear opportunities for parallelism:
  distribute “chunks” of [| 2..N |] among processors
Scalable and Robust Performance

Consider the problem of computing primality flags:

```haskell
fun isPrime i = ...

val L = [| 2..N |]

val primalityFlags = [| isPrime i | i in L |]
```

- clear opportunities for parallelism:
  - distribute “chunks” of [| 2..N |] among processors
- different executions of isPrime take different times
- (re)distributing work to idle processors takes times
The Goldilocks Problem

Goal for the scheduler

[Graph showing a curve with time on the y-axis and chunk size on the x-axis, with a peak at chunk size 1 and a goal marked as the 'Goldilocks' point, situated at the optimal chunk size where time is minimized.]
The Goldilocks Problem

Goal for the scheduler

(for a given program, input, and number of processors)
Scalable and Robust Performance

An implementation of nested data parallelism should provide both *scalable* and *robust* performance:

- **Scalable** performance:
  execution time decreases in proportion to the number of processors (for a given application and hardware platform)

- **Robust** performance:
  scalable performance across many applications and hardware platforms

Ideally, an implementation of nested data parallelism should eliminate the need for performance tuning.
Work Stealing

Busy processors “post” work to their own work queue. Idle processors “steal” work from another’s work queue.
Work Stealing

Busy processors “post” work to their own work queue. Idle processors “steal” work from another’s work queue.
Work Stealing

Busy processors “post” work to their own work queue. Idle processors “steal” work from another’s work queue.
Work Stealing

Work-first Principle:
Minimize the non-stealing costs, even at the expense of stealing costs.

- Scheduling costs = stealing costs + non-stealing costs
- Stealing (should be) rare for parallel applications:
  stealing gives processors enough work to stay busy most of the time

NOTE: non-stealing costs correspond to “posting” work that is not stolen
(so, the poster performs the work)
Parallel Sequence Representation

Manticore represents parallel sequences as balanced binary trees (with arrays of elements at leaves).

- Similar to *ropes* and *conc* lists.
- Efficient traversal, split, and concatenation.
- Dispersed in memory.
- \(O(\log n)\) random access and balance maintenance.

\[
\begin{array}{cccccccc}
| & 2 & 3 & 4 & 5 & 6 & 7 & 8 |
\end{array}
\Rightarrow
\begin{array}{cccccc}
  & 2 & & & 6 & 7 & 8 \\
  \downarrow & & & 3 & 4 & 5 \\
\end{array}
\]
Eager Tree Splitting (ETS)

Eagerly split work until a stop-splitting threshold (SST) is reached.

- Cilk++
- Intel’s Thread Building Blocks (TBB)

```haskell
fun mapETS SST f xs =
  if length xs <= SST then
    mapSeq f xs
  else let
    val (xs1, xs2) = split xs
  in
    append (\ mapETS SST f xs1,
             mapETS SST f xs2)
end
```

Performance depends on SST.
Eager Tree Splitting (ETS)

Eagerly split work until a stop-splitting threshold (SST) is reached.

- Cilk++
- Intel’s Thread Building Blocks (TBB)

```haskell
fun mapETS SST f xs =
  if length xs <= SST then
    mapSeq f xs
  else let
    val (xs1, xs2) = split xs
  in
    append (\mapETS SST f xs1,
              mapETS SST f xs2)
end
```

Performance depends on SST.
Effect of SST (48 processors)

Optimal SST depends on program, loop, data, and # procs.
Effect of SST (48 processors)

Optimal SST depends on program, loop, data, and # procs.
Lazy Tree Splitting (LTS)

Adapt Lazy Binary Splitting (Tzannes et. al. 2009) variant of work stealing to sequences implemented as balanced binary trees.

Intuition:

▸ Only profitable to “post” work if there is an idle processor.
Lazy Tree Splitting (LTS)

Adapt Lazy Binary Splitting (Tzannes et. al. 2009) variant of work stealing to sequences implemented as balanced binary trees.

Intuition:

- Only profitable to “post” work if there is an idle processor.
- Empty work-queue is a dynamic estimate of load balance. (If my work-queue transitions from non-empty to empty, then at least one processor was idle.)
Lazy Tree Splitting (LTS)

Adapt Lazy Binary Splitting (Tzannes et. al. 2009) variant of work stealing to sequences implemented as balanced binary trees.

Intuition:

- Only profitable to “post” work if there is an idle processor.
- Empty work-queue is a dynamic estimate of load balance. (If my work-queue transitions from non-empty to empty, then at least one processor was idle.)
- “Post” as much work as possible when there is an idle processor: split unprocessed elements in half; “post” one half, work on other half. (Subsequent splits with distribute work among other idle processors.)
Lazy Tree Splitting (LTS)

Adapt Lazy Binary Splitting (Tzannes et. al. 2009) variant of work stealing to sequences implemented as balanced binary trees.

Intuition:

- Only profitable to “post” work if there is an idle processor.
- Empty work-queue is a dynamic estimate of load balance. (If my work-queue transitions from non-empty to empty, then at least one processor was idle.)
- “Post” as much work as possible when there is an idle processor: split unprocessed elements in half; “post” one half, work on other half. (Subsequent splits with distribute work among other idle processors.)
- Maintain some extra bookkeeping in order to split unprocessed elements at any point in the computation. (Use zipper technique.)
Lazy Tree Splitting (LTS)

Adapt Lazy Binary Splitting (Tzannes et. al. 2009) variant of work stealing to sequences implemented as balanced binary trees.

Intuition:

- Only profitable to “post” work if there is an idle processor.
- Empty work-queue is a dynamic estimate of load balance. (If my work-queue transitions from non-empty to empty, then at least one processor was idle.)
- “Post” as much work as possible when there is an idle processor: split unprocessed elements in half; “post” one half, work on other half. (Subsequent splits with distribute work among other idle processors.)
- Maintain some extra bookkeeping in order to split unprocessed elements at any point in the computation. (Use zipper technique.)

NOTE: No magic constants!
fun isPrime i = ...

val L = [| 2, 3, 4, 5, 6, 7, 8 |]

val primalityFlags = [| isPrime i | i in L |]
Example

A

< busy >

B

Fluet

Manticore

March 30, 2012
Example

A

B

< busy >

2

4 5

3

6 7 8

3 4 5

6

7 8
Example
Example

A
< busy >
B

\[
\begin{array}{c}
\text{t} \\
\text{t} \\
4 \\
5 \\
6 \\
7 \\
8 \\
\end{array}
\]
Example

A

< idle!! >

t
f 5
6 7 8

B

Fluetic
 Mattercore
March 30, 2012 41
Example
Example

A

B

< idle!! >

t t f 5 6 7 8

Fluet

Manticore

March 30, 2012
Example
Example

```
A
```

```
B
```

```
| t | t | f |
```

```
| t | 6 |
```

```
| t | 8 |
```
Example

\begin{itemize}
\item \textbf{A} \rightarrow \textbf{B} \quad \text{< idle >}
\end{itemize}

\begin{itemize}
\item \textbf{t} \quad \textbf{t} \quad \textbf{f} \quad \textbf{t} \quad \textbf{f}
\end{itemize}
Performance Results

Machine (Dell PowerEdge R815):

- 48-cores = 4 × 12-core AMD Opteron 6172 “MagnyCours” processors
  - 2.1GHz per core
  - 64Kb L-cache, 64Kb L1-cache, & 512Kb L2-cache per core
  - 6Mb L3-cache per half-processor (six cores)
- 128Gb physical memory

Benchmarks:

- barnes-hut: 2D n-body simulation
- id-raytracer
- quicksort
- smvm: sparse matrix, (dense) vector multiplication
- dmm: dense matrix multiplication
- black-scholes: options pricing via a partial differential equation
- nested-sum: synthetic, irregular parallelism
Speedup: barnes-hut

- ETS (SST=2^{0})
- ETS (SST=2^{7})
- ETS (SST=2^{14})
- LTS

number of processors

speedup
Speedup: id-raytracer

![Graph showing speedup vs number of processors for ETS (SST=2^0), ETS (SST=2^7), ETS (SST=2^{14}), and LTS. The graph indicates a linear increase in speedup as the number of processors increases.]
Speedup: quicksort

![Graph showing speedup vs number of processors for different SST values.](image-url)
Speedup: smvm

![Graph showing speedup with number of processors for different ETS (SST) and LTS configurations.](image)
Speedup: dense-matrix-multiply

- ETS (SST=2^0)
- ETS (SST=2^7)
- ETS (SST=2^{14})
- LTS

The graph shows the speedup for different numbers of processors.
Speedup: black-scholes

The graph shows the speedup of black-scholes with different numbers of processors. The x-axis represents the number of processors, ranging from 1 to 48. The y-axis represents the speedup, ranging from 1 to 48.

The graph includes the following lines:
- Black circles: ETS (SST=2^0)
- Red triangles: ETS (SST=2^7)
- Green squares: ETS (SST=2^{14})
- Blue crosses: LTS

Each line represents a different configuration of the black-scholes algorithm, demonstrating how the speedup changes with the number of processors.
Speedup: nested-sums

- ETS (SST=2^0)
- ETS (SST=2^7)
- ETS (SST=2^{14})
- LTS

The graph shows the speedup of nested-sums with varying numbers of processors. The speedup is plotted on the y-axis, and the number of processors is plotted on the x-axis. Different lines represent different SST values, with distinct symbols for ETS and LTS.
Lazy Tree Splitting: Summary

- LTS is faster than most ETS configurations.
- In worst case, LTS time is only $1.33 \times$ best ETS time (on 48-cores).
- LTS has scalable and robust performance and requires no tuning.
  - Optimal tuning of ETS for arbitrary programs is unrealistic.
  - Static thresholds, inevitably, are a poor chunk size for some config.
- Modest implementation effort.

Future work:
- Generalize to transform an arbitrary divide-and-conquer algorithm to the corresponding LTS algorithm.
Questions?

http://manticore.cs.uchicago.edu