Manticore:
A heterogeneous parallel language

Matthew Fluet

Rochester Institute of Technology

April 26, 2012
Manticore is a *heterogeneous parallel* programming language aimed at general-purpose applications running on multi-core processors.

- hardware supports parallelism/concurrency at multiple levels
- software exhibits parallelism/concurrency at multiple levels
- to maximize productivity and performance, programming languages must support parallelism/concurrency 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
- a number of REU students — University of Chicago
- ?? YOU (an REU student) — Rochester Institute of Technology ??

and supported (in part) by the

- National Science Foundation
Digression: Parallelism vs. Concurrency

One take on this (terminology not universal, but distinction important):

Software is **parallel** if a primary intellectual challenge is using extra computational resources to do more useful work per unit time.

- Examples: scientific computing, most graphics, a lot of servers
- Key challenge is Amdahl’s Law (no sequential bottlenecks)
- Often provide parallelism via threads on different processors
- Ideally deterministic, but not when concurrent

Software is **concurrent** if a primary intellectual challenge is responding to external events from multiple sources in a timely manner.

- Examples: operating system, GUI, version control
- Key challenge is responsiveness (interactivity)
- Often provide responsiveness via threads
- Inherently non-deterministic, but not necessarily parallel
Parallelism/Concurrency in Hardware

Hardware supports parallelism/concurrency at multiple levels:

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

Software exhibits parallelism/concurrency at multiple levels. Consider a networked flight simulator:
Parallelism/Concurrency in Software

Software exhibits parallelism/concurrency 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
  - Flight Simulator

- Graphics

- SIMD parallelism for physics simulation
Parallelism/Concurrency in Software

Software exhibits parallelism/concurrency 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/Concurrency in Software

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

- parallel and/or concurrent threads for preloading terrain and computing level-of-detail refinements
Parallelism/Concurrency in Software

Software exhibits parallelism/concurrency 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**
  - speculative search for artificial intelligence

- **Graphics**
  - (visual representation)
Parallelism/Concurrency in Software

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

- concurrent 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/concurrency
  - commodity machines with multiple levels of hardware parallelism/concurrency
  - maximize productivity and performance
  - balance programmer and compiler effort

<table>
<thead>
<tr>
<th>Languages</th>
<th>Programmer Effort</th>
<th>Compiler Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>CML, Erlang, Java</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>OpenMP, NESL, Id, pH</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>SISAL</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Manticore, Parallel Haskell, Chapel, Fortress, X10</td>
<td>High</td>
<td>Low</td>
</tr>
</tbody>
</table>
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
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
  
  ```ml
  spawn exp
  ```

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

- The CML-style event combinators
  
  ```ml
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

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):

\[ [ \mid \text{exp} \mid \text{pat}_i \text{ in exp}_i \text{ where pred} \mid ] \]

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

\[ [ \mid x + y \mid x \text{ in } xs, y \text{ in } ys \mid ] \]

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

**NOTE:** no mutation, no constant-time random-access
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 Seqs: Image Manipulation

Parallel arrays are a natural representation for images:

```ml
type pixel = int * int * int
type img = pixel parray parray
```

Image transformations expressed as a computation that is applied to each pixel of an image

```ml
fun xformImg xformPix img = [
  [| [| xformPix pix | pix in row |] | row in img |]
]

fun rgbPixToGrayPix ((r, g, b) : pixel) : pixel = 
  let val m = (r + g + b) / 3
  in (m, m, m)
  end

fun rgbImgToGrayImg (img : img) : img = 
  xformImg rgbPixToGrayPix img
```
Parallel Seqs: Sparse-matrix Vector Multiplication

Parallel arrays can represent both dense and sparse vectors and matrices:

\[
\text{type vector} = \text{real parray} \\
\text{type sparse_vector} = (\text{int} \times \text{real}) \text{ parray} \\
\text{type sparse_matrix} = \text{sparse_vector parray}
\]

To multiply a sparse matrix by a dense vector, compute the dot product for each row:

\[
\text{fun dotp (sv: sparse_vector) (v: vector) : real = sumP [\| x \times (v!i) \| (i,x) \in sv ]}
\]
\[
\text{fun smvm (sm: sparse_matrix) (v: vector) : vector = [\| dotp (row, v) \| row \in sm ]}
\]
Parallel Seqs: Quicksort

Quicksort an array of integers:

```haskell
fun quicksort (a: int parray) : int parray =
  if lengthP a < 2
    then a
  else let val pivot = ns ! 0
    val ss = [\ |
              filterP cmp a
              | cmp in [\ |
                            fn x => x < pivot,
                            fn x => x = pivot,
                            fn x => x > pivot |
              |
            ] |
    val rs = [\ |
                quicksort a | a in [\ |
                                ss !0, ss !2 |
              |
            ] |
    val sorted_lt = rs !0
    val sorted_eq = ss !1
    val sorted_gt = rs !1
  in flattenP [\ | sorted_lt, sorted_eq, sorted_gt |
             ]
  end

Some awkwardness in using parallel arrays exclusively.
Parallel Tuples

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

```plaintext
datatype tree = Lf of int | ND of tree * tree

fun treeAdd (Lf n) = n
  | treeAdd (ND(t1, t2)) =
      (op +) (treeAdd t1, treeAdd t2)
```

Parallel Tuples

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

\[
\text{datatype } \text{tree} = \text{Lf of int} \mid \text{ND of tree } \ast \text{ tree}
\]

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

Quicksort an array of integers:

fun quicksort (a: int parray) : int parray =
    if lengthP a < 2
        then a
    else let val pivot = ns ! 0
        val (sorted_lt, sorted_eq, sorted_gt) = 
            (| quicksort (filterP (fn x => x < pivot) a), 
                filterP (fn x => x = pivot) a, 
                quicksort (filterP (fn x => x > pivot) a) |)
        in flattenP [| sorted_lt, sorted_eq, sorted_gt |]
    end

More natural using parallel tuples.
Parallel Bindings

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

```plaintext
fun treeMul (LF n) = n
| treeMul (ND(t1, t2)) = let
  pval 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.

fun treeMul (LF n) = n
| treeMul (ND(t1, t2)) = let
  pval b = treeMul t2
  val a = treeMul t1
  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.

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

Parallel bindings provide work-stealing parallelism.

\begin{itemize}
    \item let \texttt{dval} \( x = \text{exp}_1 \) in \text{exp}_2 end
    \item mnemonic: main thread executes \( \uparrow \text{exp}_1 \)
\end{itemize}
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:

```plaintext
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:

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

Can be used to define a number of common idioms:

\[
e_1 \: |?| \: e_2 \equiv \text{pcase} \: e_1 \: \& \: e_2
\quad \text{of} \quad n \: \& \: ? \Rightarrow n
\quad \mid \: ? \: \& \: n \Rightarrow n
\]

\[
e_1 \: |\text{andalso}| \: e_2 \equiv \ldots
\]

\[
e_1 \: |\text{orelse}| \: e_2 \equiv \ldots
\]

\[
e_1 \: |*| \: e_2 \equiv \text{pcase} \: e_1 \: \& \: e_2
\quad \text{of} \quad 0 \: \& \: ? \Rightarrow 0
\quad \mid \: ? \: \& \: 0 \Rightarrow 0
\quad \mid \: a \: \& \: b \Rightarrow a \: * \: b
\]
Can be used to define a number of common idioms:

\[
\begin{align*}
\text{e1} \mid \text{?} \mid \text{e2} & \equiv \text{pcase } \text{e1} \& \text{e2} \\
& \quad \text{of } n \& ? \Rightarrow n \\
& \quad | ? \& n \Rightarrow n \\
\text{e1} \mid \text{andalso} \mid \text{e2} & \equiv \ldots \\
\text{e1} \mid \text{orelse} \mid \text{e2} & \equiv \ldots \\
\text{e1} \mid \ast \mid \text{e2} & \equiv \text{pcase } \text{e1} \& \text{e2} \\
& \quad \text{of } 0 \& ? \Rightarrow 0 \\
& \quad | ? \& 0 \Rightarrow 0 \\
& \quad | a \& b \Rightarrow a \ast b
\end{align*}
\]

\[
\begin{align*}
\text{fun} \ \text{treeMul} \ (\text{LF } n) & = n \\
& | \ \text{treeMul} \ (\text{ND}(t1, t2)) = \\
& \quad \text{treeMul} \ t1 \mid \ast \mid \text{treeMul} \ t2
\end{align*}
\]
Exceptions

Exceptions follow a *sequential semantics*.

\[
\begin{align*}
\text{fun } f \ n &= \text{if } n = 100 \text{ then raise Foo else } \ldots \\
\text{fun } g \ n &= \text{if } n = 100 \text{ then raise Goo else } \ldots \\
(\mid f\ 100,\ g\ 100\mid) \\
\text{handle } Foo &= \ldots \quad (* \text{handled } Foo *) \\
\mid Goo &= \ldots \quad (* \text{unreachable} *)
\end{align*}
\]
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

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 *)
```

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

Requires a slightly more restrictive implementation of the implicitly-threaded parallel constructs, but the precise semantics is crucial for systems programming.
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.

```haskell
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:

```haskell
fun isPrime i = ...

val L = [| 2..N |]

val primalityFlags = [| isPrime i | i in L |]
```
Scalable and Robust Performance

Consider the problem of computing primality flags:

```plaintext
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:

```ml
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

Goal for the scheduler (for a given program, input, and number of processors)

time

chunk size

whole problem
The Goldilocks Problem

For a given program, input, and number of processors, the goal for the scheduler is to find a chunk size that minimizes time. This is represented by the point where the curve is at its lowest point, indicating an optimal balance between chunk size and time. The diagram shows the trade-off between chunk size and whole problem time, with the optimal chunk size located at the bottom of the curve.

(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.

\[
[1, 2, 3, 4, 5, 6, 7, 8] \Rightarrow
\]

```
[1 2, 3, 4, 5, 6, 7, 8]  \Rightarrow
```

```
2
3 4 5
6
7 8
```
Eager Tree Splitting (ETS)

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

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

```reason
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)

```
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

\[
A \xrightarrow{} B
\]

\[
\begin{array}{c}
A \\
\quad \\
\quad \\
\quad 2
\end{array}
\]

\[
\begin{array}{c}
B \\
\quad \\
\quad < busy >
\end{array}
\]

\[
\begin{array}{c}
\quad \\
\quad \\
\quad \\
\quad 3
\end{array}
\]

\[
\begin{array}{c}
\quad \\
\quad \\
\quad \\
\quad 4 \quad 5
\end{array}
\]

\[
\begin{array}{c}
\quad \\
\quad \\
\quad \\
\quad 6 \quad 7 \quad 8
\end{array}
\]
Example
Example
Example
Example
Example

```
A
< idle!! >
```

```
B
```

```
t t f 5 6 7 8
```

```
Fluet
Manticore
April 26, 2012
```
Example

A

< idle!! >

t t f 5 6 7 8

B

< idle!! >
Example
Example
Example
Performance Results

Machine (Dell PowerEdge R815):

- cadmium.cs.rit.edu
- 48-cores = 4 × 12-core AMD Opteron 6172 “MagnyCours” processors
  - 2.1GHz per core
  - 64Kb I-cache, 64Kb L1-cache, & 512Kb L2-cache per core
  - 6Mb L3-cache per half-processor (six cores)
- 128Gb physical memory
- only $10,444.84 (in late 2010)
Performance Results

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
Speedup: id-raytracer

![Graph showing speedup vs number of processors for different SST values. The graph includes three lines labeled ETS (SST=2^0), ETS (SST=2^7), and ETS (SST=2^{14}), with the LTS line also present. The x-axis represents the number of processors, ranging from 1 to 48, and the y-axis represents the speedup, ranging from 1 to 48. Each data point corresponds to a specific number of processors and is marked with either a black dot, red triangle, or green cross, indicating the speedup achieved by each SST variant.]
Speedup: quicksort

![Graph showing speedup with different processors and SST values.](image)
Speedup: smvm

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

Number of processors vs. speedup

ETS (SST=2^0) has the highest speedup, followed by ETS (SST=2^7) and then ETS (SST=2^{14}). LTS has the lowest speedup.
Speedup: dense-matrix-multiply

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

(number of processors vs. speedup graph)
Speedup: black-scholes

![Graph showing speedup vs number of processors for different SST values.]

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

The graph illustrates the speedup achieved with increasing numbers of processors for various SST configurations.
Speedup: nested-sums

The diagram shows the speedup of different systems as a function of the number of processors. The systems are ETS (SST=2^0), ETS (SST=2^7), ETS (SST=2^14), and LTS. The speedup is plotted on the y-axis, and the number of processors is on the x-axis. The graph indicates that the speedup increases with the number of processors for all systems, but the performance varies depending on the specific system and configuration.
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