The Manticore Project
(a status report)

John Reppy

University of Chicago

September 13, 2012
People

The Manticore project is a joint project between the University of Chicago and the Rochester Institute of Technology.

Carsen Berger  University of Chicago  
Lars Bergstrom  University of Chicago  
Matthew Fluet  Rochester Institute of Technology  
Mike Rainey  Max Plank Institute  
Stephen Rosen  University of Chicago  
Nora Sandler  University of Chicago  
Adam Shaw  University of Chicago
Background and motivation

- Well-known sea change in the design of microprocessors.
- Hardware supports parallelism at multiple levels: SIMD, SMT, multicore, and small-scale SMP.
- Likewise, many commodity applications exhibit parallelism at multiple levels.
- For applications to take advantage of future CPU improvements they need to be parallel.
- The Manticore project is our effort to address the programming needs of commodity applications running on the commodity parallel hardware.
Talk overview

This talk is divided into three parts:

1. Overview of Parallel ML.
2. Current status.
3. Future work.

Please see http://manticore.cs.uchicago.edu for papers.
Language Overview
The initial design of PML is purposefully conservative [ICFP ’08; JFP ’10] It can be summarized as the combination of three distinct sub-languages:

- A “pure” (mutation-free) subset of SML.
- Language mechanisms for implicitly-threaded parallel programming.
- Language mechanisms for explicitly-threaded parallel programming (a.k.a. concurrent programming).
Language design (*continued ...*)

PML provides several light-weight syntactic forms for introducing parallel computation.

- **Parallel tuples** provide a basic fork-join parallel computation.
- **Parallel bindings** provide data-flow parallelism with cancelation of unused subcomputations.
- **Parallel arrays** provide fine-grain data-parallel computations over sequences.
- **Parallel case** provides non-deterministic speculative parallelism.

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

Parallel tuples provide fork-join parallelism. For example, consider summing the leaves of a binary tree.

```plaintext
datatype tree = LF of long | ND of tree * tree

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

Diagram: See image for visual representation of parallel tuple operations.
Parallel bindings provide more flexibility than parallel tuples. For example, consider computing the product of 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
```

**NOTE:** the computation of `b` is speculative.
Parallel bindings (continued ...)

[Diagram of parallel bindings]
Parallel case

Parallel case supports speculative parallelism when we want the quickest answer (e.g., search problems). For example, consider picking a leaf of the tree:

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

There is some similarity with join patterns.
Parallel case *(continued ...)*
Nested data parallelism (NDP)

We support fine-grained, nested data-parallel computation using a parallel array comprehension form (NESL/Nepal/DPH):

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

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

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

**NOTE:** zip semantics, not Cartesian-product semantics.
Nested data parallelism (*continued ...*)

Mandelbrot set computation:

```latex
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;

[| [|| loop(0, x i, y j) | i in [| 0..N ||] ||]
  | j in [| 0..N ||]
|]
```
Explicit threading

Based on Concurrent ML design.

- Explicit threading with preemptive scheduling. Threads may contain implicitly-threaded parallelism.
- Threads do not share state; instead they communicate and synchronize via message passing.
- Synchronization and communication abstractions are supported by the mechanism of first-class synchronous operations (called events) [PLDI ’91’].
- In addition to supporting abstraction, events also provide a uniform framework for synchronous system interfaces (e.g., I/O).
Current Status
Implementation highlights

- *De novo* implementation.
- Efficient work-stealing techniques for managing parallelism [ICFP ’10; JFP ’12; Rainey’s PhD 2010].
- Use ropes to implement NDP [Shaw’s MS ’07; ICFP ’11].
- New protocol for CML on SMP [ICFP ’09].
- NUMA aware parallel GC [MSPC ’11].
Memory model

GC is a combination of the Appel Semi-generational collector and the Doligez-Leroy-Gonthier parallel collector.

- Minor GCs are completely asynchronous.
- Major GC are mostly asynchronous.
- NUMA-aware global GCs are parallel stop-the-world.
Performance

![Graph showing performance vs. number of processors]

- Perfect-Speedup
- Ray-Tracer
- Black-Scholes
- Quicksort
- Barnes-Hut
Future Work
Data-only flattening

- Traditional NDP implementation techniques are based on flattening (aka vectorization).
- Manticore does not flatten NDP, but instead relies on efficient work-stealing to get NDP speedups.
- **Observation:** flattening improves data access and allows SIMD instructions, but distorts the control-flow of the program.
Data-only flattening (continued ...)

Mandlebrot

SxVM
Adding shared state to PML

- Shared state can provide asymptotic performance improvements (e.g., transposition tables in search).
- Shared mutable data structures lead to clearer algorithms (e.g., mesh refinement).
- **Observation:** mutable shared memory is a hardware accelerated broadcast mechanism.
Adding shared state to PML (continued ...)
We have a number of projects to improve sequential performance and robustness:

- Shao-Appel closure conversion.
- Reflow analysis for better inlining.
- Switching to a modified version of MLton’s front end.
- Provide complete Basis Library
- Considering possible LLVM back end.
Final comments

- Strict, pure FP is a really good base for parallelism.
- State is evil (but necessary).
- It is better to build for parallelism from the start.

http://manticore.cs.uchicago.edu