Implicit threading, Speculation, and Mutation in Manticore

John Reppy

University of Chicago

April 30, 2010

Overview

This talk is about some very preliminary ideas for the next step in the design of the Parallel ML (PML) language, which is part of the Manticore system.
Specifically, we are exploring three related ways to grow the language:

1. better support for speculative parallelism
2. support for shared state between implicitly-parallel computations
3. supporting nondeterminism to increase parallelism

This talk focuses on the first two.

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project

- The Manticore project is our effort to address the programming needs of commodity applications running on multicore SMP systems
- Prototype language design supports different levels of parallelism
- High-level declarative mechanisms for fine-grain parallelism
- No shared memory
- Preserve determinism where possible

The Manticore Project *(continued ...)*

Our initial language design is called Parallel ML (PML).

- Sequential core language based on subset of SML: strict with no mutable storage.
- A variety of lightweight implicitly-threaded constructs for fine-grain parallelism.
- Explicitly-threaded parallelism based on CML: message passing with first-class synchronization.
- Prototype implementation with good scaling on 16-way parallel hardware for a range of applications.

The Manticore Project *(continued ...)*

Our initial language design is called Parallel ML (PML).

- Sequential core language based on subset of SML: strict with no mutable storage.
- A variety of lightweight implicitly-threaded constructs for fine-grain parallelism.
- Explicitly-threaded parallelism based on CML: message passing with first-class synchronization.
- Prototype implementation with good scaling on 16-way parallel hardware for a range of applications.

The Manticore Project *(continued ...)*

Our initial language design is called Parallel ML (PML).

- Sequential core language based on subset of SML: strict with no mutable storage.
- A variety of lightweight implicitly-threaded constructs for fine-grain parallelism.
- Explicitly-threaded parallelism based on CML: message passing with first-class synchronization.
- Prototype implementation with good scaling on 16-way parallel hardware for a range of applications.

The Manticore Project *(continued ...)*

Our initial language design is called Parallel ML (PML).

- Sequential core language based on subset of SML: strict with no mutable storage.
- A variety of lightweight implicitly-threaded constructs for fine-grain parallelism.
- Explicitly-threaded parallelism based on CML: message passing with first-class synchronization.
- Prototype implementation with good scaling on 16-way parallel hardware for a range of applications.

Implicit threading

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

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

Implicit threading

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

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

Implicit threading

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

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

Implicit threading

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

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

Implicit threading

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

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

Implicit threading

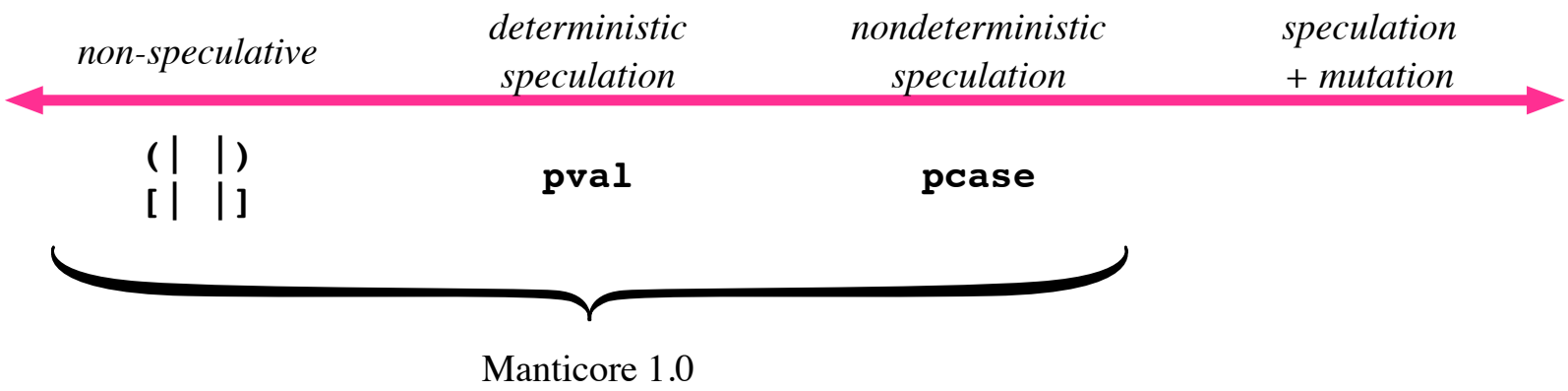
PML provides several light-weight syntactic forms for introducing parallel computation.

- Nested Data-parallel arrays provide fine-grain data-parallel computations over sequences.
- Parallel tuples provide a basic fork-join parallel computation.
- Parallel bindings provide data-flow parallelism with cancellation of unused subcomputations.
- Parallel cases provide non-deterministic speculative parallelism.

These forms are declarations that a computation is a good candidate for parallel execution, but the details are left to the implementation.

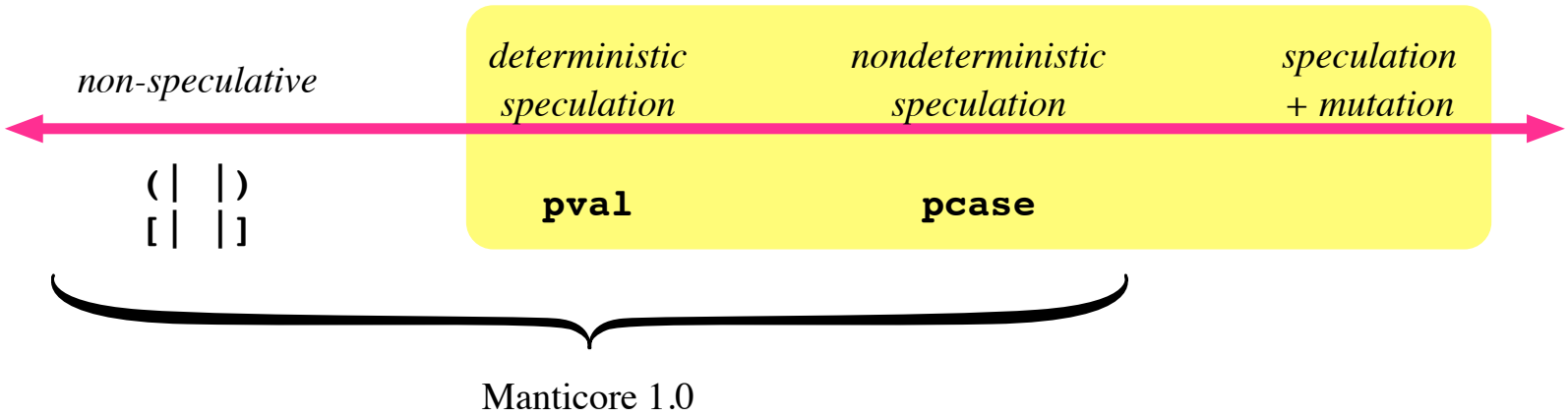
Implicit threading design space

We can organize these features by how well behaved they are.



Implicit threading design space

We can organize these features by how well behaved they are.



The need for speculation

- Amdahl's Law tells us that as the number of cores increases, execution time will be dominated by sequential code.
- Speculation is an important tool for introducing parallelism in otherwise sequential code.
- PML supports both deterministic and nondeterministic speculation.

The need for speculation

- Amdahl's Law tells us that as the number of cores increases, execution time will be dominated by sequential code.
- Speculation is an important tool for introducing parallelism in otherwise sequential code.
- PML supports both deterministic and nondeterministic speculation.

The need for speculation

- Amdahl's Law tells us that as the number of cores increases, execution time will be dominated by sequential code.
- Speculation is an important tool for introducing parallelism in otherwise sequential code.
- PML supports both deterministic and nondeterministic speculation.

Parallel bindings

Parallel bindings allow deterministic speculation. For example, the computation of `b` may not be needed in the following program:

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

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

```
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.

Extending speculation to collections

Using **pcase**, one can define speculative functions over lists:

```
fun pExists f [] = false
  | pExists f (x::xs) = (pcase f x & pExists f xs
     of true & ? => true
      | ? & true => true
      | false & false => false)
```

We plan to provide similar operations on our data-parallel arrays, which will avoid the sequential bottleneck of the list-based operations.

The need for shared mutable state

- Mutable storage is a very powerful communication mechanism: essentially a broadcast mechanism supported by the memory hardware.
- Sequential algorithms and data-structures gain significant (asymptotic) performance benefits from shared memory (*e.g.*, union-find with path compression).
- Some algorithms seem hard/impossible to parallelize without shared state (e.g., mesh refinement).
- But shared memory makes parallel programming hard, so we want to be cautious in adding it to PML.

The need for shared mutable state

- Mutable storage is a very powerful communication mechanism: essentially a broadcast mechanism supported by the memory hardware.
- Sequential algorithms and data-structures gain significant (asymptotic) performance benefits from shared memory (*e.g.*, union-find with path compression).
- Some algorithms seem hard/impossible to parallelize without shared state (e.g., mesh refinement).
- But shared memory makes parallel programming hard, so we want to be cautious in adding it to PML.

The need for shared mutable state

- Mutable storage is a very powerful communication mechanism: essentially a broadcast mechanism supported by the memory hardware.
- Sequential algorithms and data-structures gain significant (asymptotic) performance benefits from shared memory (*e.g.*, union-find with path compression).
- Some algorithms seem hard/impossible to parallelize without shared state (e.g., mesh refinement).
- But shared memory makes parallel programming hard, so we want to be cautious in adding it to PML.

The need for shared mutable state

- Mutable storage is a very powerful communication mechanism: essentially a broadcast mechanism supported by the memory hardware.
- Sequential algorithms and data-structures gain significant (asymptotic) performance benefits from shared memory (*e.g.*, union-find with path compression).
- Some algorithms seem hard/impossible to parallelize without shared state (e.g., mesh refinement).
- But shared memory makes parallel programming hard, so we want to be cautious in adding it to PML.

The design challenge

- How do we add shared memory while preserving PML's declarative programming model for fine-grain parallelism?
- Some races are okay in an implicitly-threaded setting.
- Deadlock is not okay in an implicitly-threaded setting.

The design challenge

- How do we add shared memory while preserving PML's declarative programming model for fine-grain parallelism?
- Some races are okay in an implicitly-threaded setting.
- Deadlock is not okay in an implicitly-threaded setting.

The design challenge

- How do we add shared memory while preserving PML's declarative programming model for fine-grain parallelism?
- Some races are okay in an implicitly-threaded setting.
- Deadlock is not okay in an implicitly-threaded setting.

Example: Minimax search

- Algorithms, such as minimax search and SAT solving, can benefit from sharing information between branches of the search.
- For game search, transposition tables are used to turn the search tree into a DAG.
- Transposition tables provide a kind of function memoization.

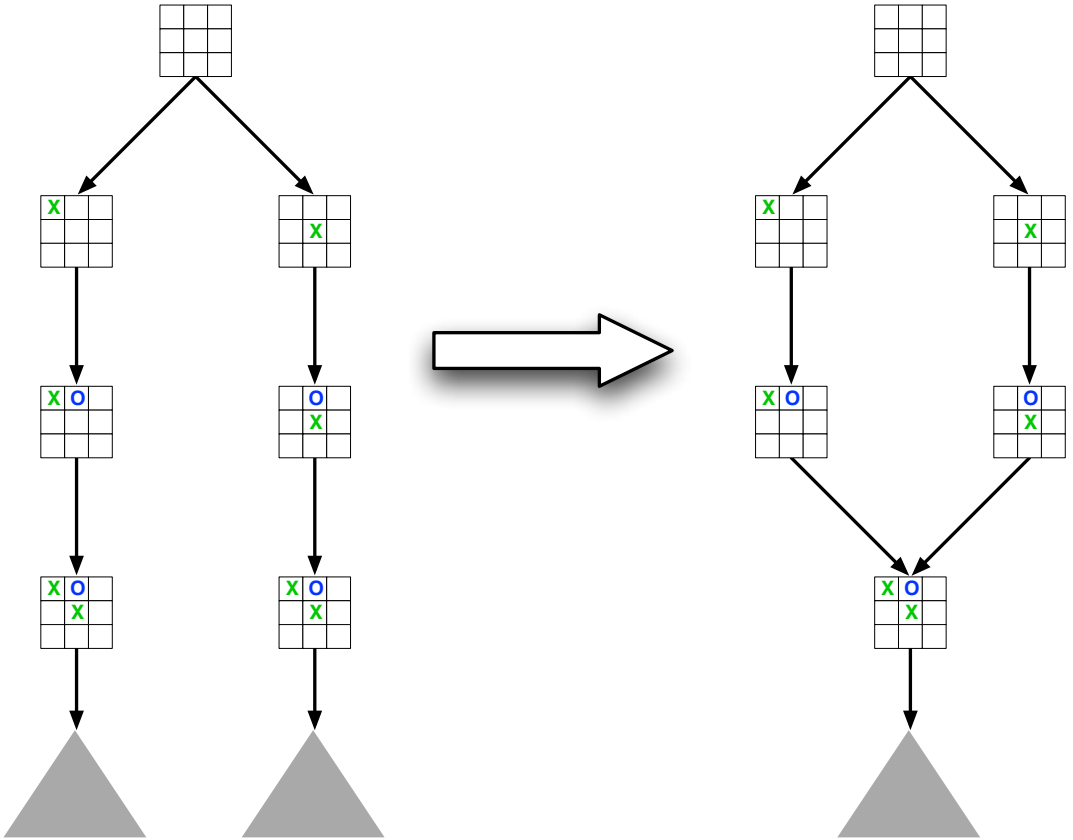
Example: Minimax search

- Algorithms, such as minimax search and SAT solving, can benefit from sharing information between branches of the search.
- For game search, transposition tables are used to turn the search tree into a DAG.
- Transposition tables provide a kind of function memoization.

Example: Minimax search

- Algorithms, such as minimax search and SAT solving, can benefit from sharing information between branches of the search.
- For game search, transposition tables are used to turn the search tree into a DAG.
- Transposition tables provide a kind of function memoization.

Transposition tables



Transposition tables *(continued ...)*

- For Tic-Tac-Toe, using a transposition table results in a 34-fold reduction in the number of board positions searched.
- Thus, a sequential search that uses a transposition table is likely to beat a parallel search that doesn't.
- Want to support sharing information between parallel branches.

Transposition tables *(continued ...)*

- For Tic-Tac-Toe, using a transposition table results in a 34-fold reduction in the number of board positions searched.
- Thus, a sequential search that uses a transposition table is likely to beat a parallel search that doesn't.
- Want to support sharing information between parallel branches.

Transposition tables *(continued ...)*

- For Tic-Tac-Toe, using a transposition table results in a 34-fold reduction in the number of board positions searched.
- Thus, a sequential search that uses a transposition table is likely to beat a parallel search that doesn't.
- Want to support sharing information between parallel branches.

Possible feature: Blackboards

- Implementing memoization in a general-purpose language poses a lot of challenges, such as efficient equality testing and hashing for user-defined types.
- An alternative are blackboards, which are essentially shared hash tables.

Possible feature: Blackboards

- Implementing memoization in a general-purpose language poses a lot of challenges, such as efficient equality testing and hashing for user-defined types.
- An alternative are blackboards, which are essentially shared hash tables.

Adding blackboards to PML

Write-once blackboards; returns value if already set.

```
val setIfNotSet : ('a bb * key * 'a) -> 'a option
```

Memoize a function so that it is evaluated no more than once per call.

```
fun memo (bb, keyFn, f) x = let
      val iv = ivarNew()
      in
        case setIfNotSet (bb, keyFn x, iv)
         of SOME iv' => ivarGet iv'
          | NONE => let
              val y = f x
              in ivarSet (iv, y); y end
      end
```

This approach fits well with our existing mechanisms.

What is the right level of abstraction?

- One approach is to provide a collection of parallel data structures, such as blackboards.
- Alternatively, we can provide lower-level mechanisms.
- For example, blackboards can be built from synchronous variables (MVars and IVars), but synchronous variables introduce the possibility of deadlock.
- Perhaps STM is the solution?

What is the right level of abstraction?

- One approach is to provide a collection of parallel data structures, such as blackboards.
- Alternatively, we can provide lower-level mechanisms.
- For example, blackboards can be built from synchronous variables (MVars and IVars), but synchronous variables introduce the possibility of deadlock.
- Perhaps STM is the solution?

What is the right level of abstraction?

- One approach is to provide a collection of parallel data structures, such as blackboards.
- Alternatively, we can provide lower-level mechanisms.
- For example, blackboards can be built from synchronous variables (MVars and IVars), but synchronous variables introduce the possibility of deadlock.
- Perhaps STM is the solution?

What is the right level of abstraction?

- One approach is to provide a collection of parallel data structures, such as blackboards.
- Alternatively, we can provide lower-level mechanisms.
- For example, blackboards can be built from synchronous variables (MVars and IVars), but synchronous variables introduce the possibility of deadlock.
- Perhaps STM is the solution?

People

Lars Bergstrom	*University of Chicago*
Matthew Fluet	*Rochester Institute of Technology*
Mike Rainey	*University of Chicago*
Adam Shaw	*University of Chicago*
Yingqi Xiao	*University of Chicago*

with help from

Nic Ford	*University of Chicago*
Korie Klein	*University of Chicago*
Joshua Knox	*University of Chicago*
Jon Riehl	*University of Chicago*
Ridg Scott	*University of Chicago*

`http://manticore.cs.uchicago.edu`