The Manticore Project

Matthew Fluet
mtf@cs.rit.edu

Computer Science Department
Rochester Institute of Technology

September 23, 2013
Functional High-Performance Computing (FHPC’13)
The Manticore Project

An effort to design and implement a parallel functional programming language aimed at general-purpose applications running on multi-core processors:

- commodity applications with multiple levels of software parallelism
- commodity hardware with multiple levels of hardware parallelism

Programming language must support parallelism at multiple levels. We call this property *heterogeneous parallelism*.

- maximize productivity and performance
- balance programmer and compiler effort
Outline

- Overview of Manticore
  - Manticore – the Project and Vision
  - Manticore – the Language
  - Manticore – the Implementation
- Past Work
- Present Work
  - NUMA aware garbage collection
  - Data-only flattening for nested-data parallelism
  - Controlled mutable state
- Conclusion
Overview of Manticore
The Manticore Project is a joint project between researchers at the University of Chicago and the Rochester Institute of Technology:

- Lars Bergstrom — Mozilla Research (prev. UChicago)
- Matthew Fluet — Rochester Institute of Technology
- Matthew Le — Rochester Institute of Technology
- Mike Rainey — Max Planck Institute for Software Systems (prev. UChicago)
- John Reppy — University of Chicago
- Adam Shaw — University of Chicago
- A number of REU students — UChicago and RIT

and supported (in part) by the

- National Science Foundation

Happy Birthday! Source repository initial commit: 2006-09-19
Hardware supports parallelism at multiple levels:

- single instruction, multiple data (SIMD) instructions

- simultaneous multithreading executions

- multicore processors

- multiprocessor systems
Software exhibits parallelism at multiple levels. Consider a networked flight simulator:
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**: Flight Simulator
- **Graphics**:

- **SIMD parallelism for physics simulation**
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 (e.g., rain, fog, clouds)
- Artificial Intelligence: Flight Simulator
- Graphics: data-parallel computations to model natural phenomena (e.g., rain, fog, and clouds)
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**
  - Flight Simulator

- **Graphics**

- parallel threads for preloading terrain
  and computing level-of-detail refinements
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:** speculative search for artificial intelligence
- **Graphics:**
Software exhibits parallelism at multiple levels.
Consider a networked flight simulator:

- parallel threads for user interface and network components
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 – The Project

A long-range project with two major aspects:

- Language design for heterogeneous parallel programming.
- Language implementation for heterogeneous parallelism.
Manticore – The Language

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
The Language: 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
The Language: 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
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)
The Language: 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
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

*Sequential and deterministic semantics (mostly)*: simplify programming

*Cancellation*: unused/abandoned subcomputations

(see ICFP’08a, JFP’11)
Manticore – The Implementation

Implementation provided by three primary components:

- **Compiler**
  - compile source Manticore programs to x86-64 executables

- **Initial Basis Library**
  - various libraries included when compiling Manticore programs

- **Runtime System**
  - process abstractions and garbage collection services

(and many unfortunate, non-modular dependencies between them)
The compiler is organized as a series on transformations between IRs:

- **Typed AST** — explicitly-typed, polymorphic, abstract-syntax tree.
- **BOM** — direct-style, normalized $\lambda$-calculus w/ continuation primitives.
- **CPS** — continuation-passing-style $\lambda$-calculus.
- **CFG** — first-order control-flow graph
The Implementation: Compiler

AST transformations:
- Desugaring syntax and compiling pattern matching.
- Introducing futures for implicitly-threaded parallel features.
- Flattening of nested-data-parallelism (AOS to SOA).

BOM/CPS transformations:
- Standard functional-compiler optimizations
  (e.g., arity-raising, reflow-based inlining, contraction, ...).

Codegen
- Target x86-64 architecture with MLRISC framework.
The Implementation: Initial Basis Library

Utility Libraries:
- integers, doubles, strings, lists, ...

Support Libraries:
- ropes, synchronized data structures, scheduler actions, ...

Inline BOM:
- BOM IR has a concrete syntax.
- BOM code can be embedded in Manticore source code.
- Used to implement concurrency and parallel features.
- Used to import scheduler code and implement scheduler operations.
Distinguish *computation*

- **Fibers**
  - fundamental unit of (sequential) control
  - correspond to an active or a suspended computation
  - represented as a heap-allocated first-class continuation
  - *fiber-local storage (FLS)* provides per-fiber environment

- **Threads:**
  - correspond to language-level threads with an ID
  - represented as a fiber with FLS-maintained ID

from *computational resource*:

- **Virtual Processors (VProcs):**
  - abstraction of a computational resource
  - primary ready queue of fibers (local access only)
  - landing pad (global access, lock-free stack) for incoming fibers

- **Processor Topology**
The Implementation: Runtime System

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.

- Invariant: no pointers from global heap to local heaps
- Invariant: no pointers from one local heap to another
- Minor GCs are completely asynchronous
- Major GCs are mostly asynchronous
- NUMA-aware global GCs are parallel stop-the-world

- Objects shared between VProcs must be promoted to the global heap.
Manticore: The Performance

![Graph showing speedup vs. sequential for different benchmarks across varying numbers of processors.]

- Perfect-Speedup
- Ray-Tracer
- Black-Scholes
- Quicksort
- Barnes-Hut

The Manticore Project
Past Work
Past Work

- a parallel implementation of Concurrent ML (see POPL’07, DAMP’08, & ICFP’09)

- a novel infrastructure for nested schedulers (see ICFP’08b, Rainey PhD)

- a collection of expressive implicitly-threaded parallel constructs with mostly sequential semantics and future-based implementations (see ICFP’08a, JFP’11, Shaw PhD)

- a Lazy Tree Splitting (LTS) strategy for performance-robust work-stealing of parallel computations over irregular tree-like data structures (see ICFP’10, JFP’12, Rainey PhD)
Past Work

- a parallel implementation of Concurrent ML (see POPL’07, DAMP’08, & ICFP’09)
- a novel infrastructure for nested schedulers (see ICFP’08b, Rainey PhD)
- a collection of expressive implicitly-threaded parallel constructs with mostly sequential semantics and future-based implementations (see ICFP’08a, JFP’11, Shaw PhD)
- a Lazy Tree Splitting (LTS) strategy for performance-robust work-stealing of parallel computations over irregular tree-like data structures (see ICFP’10, JFP’12, Rainey PhD)
Nested Schedulers
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
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

Infrastructure highlights:
- A *scheduler action* is a function that implements scheduling logic (e.g., context switching) for an individual VProc.
- A VProc has a stack of scheduler actions.
- A scheduler action performs scheduler specific duties and concludes either by forwarding a preempted fiber up the stack or by pushing a new scheduler onto the stack and running a fiber.
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

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.
- expressing (and achieving) ideal/fair behavior for heterogeneous parallelism; look to systems community and OS scheduling.
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

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.
- expressing (and achieving) ideal/fair behavior for heterogeneous parallelism; look to systems community and OS scheduling
Lazy Tree Splitting
Lazy Tree Splitting: The “Goldilocks” Problem

Consider mapping computation over a large data structure:
- parallelize by distributing “chunks” of data structure among processors
- execution of computation on different elements takes different times
- (re)distributing work to idle processors takes times
- optimize for a given program, input, and number of processors

Eager Tree Splitting (ETS):
Eagerly split data until a stop-splitting threshold (SST) is reached.
Lazy Tree Splitting: The “Goldilocks” Problem

Consider mapping computation over a large data structure:
- parallelize by distributing “chunks” of data structure among processors
- execution of computation on different elements takes different times
- (re)distributing work to idle processors takes times
- optimize for a given program, input, and number of processors

Eager Tree Splitting (ETS):
Eagerly split data until a stop-splitting threshold (SST) is reached.
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 (and maybe yet another is 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 will 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!*
Lazy Tree Splitting: The Performance (barnes-hut)

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

The Manticore Project
Lazy Tree Splitting: The Performance (id-raytracer)

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

Graph showing speedup vs. number of processors.
Lazy Tree Splitting: The Performance (quicksort)

The Manticore Project

Number of processors vs. speedup for different ETS (ETS=SST=2^0, ETS=SST=2^7, ETS=SST=2^{14}) and LTS configurations.
Lazy Tree Splitting: The Performance (smvm)

The figure shows the speedup of different tree splitting methods as a function of the number of processors. The methods compared are ETS (SST = $2^0$), ETS (SST = $2^7$), ETS (SST = $2^{14}$), and LTS. The graphs illustrate how speedup increases with the number of processors, with ETS (SST = $2^{14}$) showing the highest speedup followed by ETS (SST = $2^7$), ETS (SST = $2^0$), and finally LTS.
Lazy Tree Splitting: The Performance (dense-matrix-multiply)

The diagram shows the performance of different tree splitting algorithms as a function of the number of processors. The algorithms compared include:

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

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 illustrates how the speedup increases with the number of processors for each of the algorithms.
Lazy Tree Splitting: The Performance (black-scholes)

The Manticore Project

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

speedup

number of processors
Lazy Tree Splitting: The Performance (nested-sums)

The performance of different splitting methods (ETS and LTS) with varying numbers of processors is depicted in the graph. The y-axis represents the speedup, and the x-axis shows the number of processors.

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

The graph shows that as the number of processors increases, the speedup generally increases as well, with some methods experiencing a more pronounced increase in performance with higher processor counts.
Present Work
Present Work

- garbage collection for multicore NUMA systems
  (see MSPC’11)

- data-only flattening for nested-data parallelism
  (see PPoPP’13, Shaw PhD)

- controlled mutable state
  (see Bergstrom PhD)
  - memoization of pure functions
    using a dynamically-sized, parallel hash table
  - automatic transactions for parallel operations
    preserving linearizability and local reasoning
Commodity systems have a non-uniform memory architecture (NUMA). While all memory is equally addressable, not all memory is equally close to a CPU.

- Remote memory accesses have higher latency
- Interconnects between packages saturate, reducing bandwidth and increasing latency

Garbage collectors and parallel code can saturate these interconnects. We show:

- Careful page allocation increases performance by roughly 7%
- Even with good locality and page allocation, may need to distribute or replicate high-contention data
Multicore NUMA Machines

AMD Opteron $\times 4 = 48$ cores:

$\begin{align*}
\text{RAM} & \rightarrow \text{Node 0, 6 cores} \rightarrow \text{Processor 0} \\
\text{RAM} & \rightarrow \text{Node 1, 6 cores} \\
\text{DDR3 Dual-Channel} & \\
\text{x16 HT3 to I/O} & \\
x8 HT3 & \\
x8 HT3 &
\end{align*}$

$x8 \ HT3 = 6.4 \ GB/s$

$\begin{align*}
\text{DDR3 1333 MHz} & = 21.3 \ GB/s
\end{align*}$
Multicore NUMA Machines

Intel Xeon × 4 = 32 cores

QPI = 25.6 GB/s
DDR3 1066 MHz = 17.1 GB/s
NUMA Allocation Policies

Physical location of a newly mapped memory page is determined when the page is first touched based on the policy.

- Same-thread (default)
- Preferred single location
- Interleaved/Round-robin
Performance of NUMA Allocation Policies

AMD; same-thread

![Graph showing performance of NUMA allocation policies with threads and speedup on the x and y axes, respectively. The graph includes lines for Ideal Speedup, Dense-Matrix-Multiply, Raytracer, Quicksort, Barnes-Hut, and SMVM. The x-axis ranges from 1 to 48 threads, and the y-axis ranges from 0 to 40 speedup.]
Performance of NUMA Allocation Policies

AMD; preferred node 0

- Default behavior with a single-threaded allocator
- Scalability to 12 cores is good: false sense of scalability
Performance of NUMA Allocation Policies

AMD; interleaved/round-robin

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Vs. baseline at 48 cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMM</td>
<td>-1.3%</td>
</tr>
<tr>
<td>Raytracer</td>
<td>0%</td>
</tr>
<tr>
<td>Quicksort</td>
<td>-7.2%</td>
</tr>
<tr>
<td>Barnes-hut</td>
<td>-7.3%</td>
</tr>
<tr>
<td>SMVM</td>
<td>50%</td>
</tr>
</tbody>
</table>

- **Worse unless** there is a piece of high-contention data.
Node-Balanced Global Garbage Collection

Per-VProc minor and major GCs preserve NUMA locality.

Parallel Copying Global GCs:
- Uneven allocation patterns by threads may lead to imbalance in per-VProc reachable objects.
- VProc copies a discovered object to a to-space chunk allocated by (and near to) the VProc.
- VProc scans its own to-space chunks.
- After exhausting its own to-space chunks, a VProc can either:
  - wait (unbalanced GC)
  - steal a to-space chunk of a node-local VProc (node-balanced GC)
  - steal a to-space chunk of any VProc (balanced GC)
Node-Balanced Global Garbage Collection

Per-VProc minor and major GCs preserve NUMA locality.

Parallel Copying Global GCs:
- After exhausting its own to-space chunks, a VProc can either:
  - wait (unbalanced GC)
  - steal a to-space chunk of a node-local VProc (node-balanced GC)
  - steal a to-space chunk of any VProc (balanced GC)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Unbalanced</th>
<th>AMD at 48 cores</th>
<th>Node-Balanced</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Global (s)</td>
<td>Total (s)</td>
<td>Global (s)</td>
</tr>
<tr>
<td>Barnes-hut</td>
<td>0.308</td>
<td>2.52</td>
<td>0.255</td>
</tr>
<tr>
<td>Quicksort</td>
<td>0.321</td>
<td>2.16</td>
<td>0.268</td>
</tr>
</tbody>
</table>
Data-Only Flattening for Nested-Data Parallelism
Data-parallelism: operating on elements of a data structure in parallel

- Flat-data parallelism: operation cannot contain data-parallelism; balanced work and good match for SIMD architectures; very effective for many regular-parallel applications;

- Nested-data parallelism: operation can contain data-parallelism; well-suited to expressing irregular-parallel applications; imbalanced work and poor match for SIMD architectures
Data-Only Flattening for Nested-Data Parallelism

Data-parallelism: operating on elements of a data structure in parallel
- Flat-data parallelism: operation cannot contain data-parallelism; balanced work and good match for SIMD architectures; very effective for many regular-parallel applications;
- Nested-data parallelism: operation can contain data-parallelism; well-suited to expressing irregular-parallel applications; imbalanced work and poor match for SIMD architectures

Blelloch introduced flattening/vectorization to compile nested-data parallelism into flat-data parallelism.
- Blelloch & Sabot: first-order NDP
- Keller, Chakravarty, & Leshchinskiy: higher-order NDP

Transforms both *code* and *data* (and all of each): replicate some computation in order to operate on more data at once.
Data-parallelism: operating on elements of a data structure in parallel

- Flat-data parallelism: operation cannot contain data-parallelism; balanced work and good match for SIMD architectures; very effective for many regular-parallel applications;

- Nested-data parallelism: operation can contain data-parallelism; well-suited to expressing irregular-parallel applications; imbalanced work and poor match for SIMD architectures

Compile nested-data parallelism to fork-join parallelism and rely on work-stealing techniques to handle load balancing (but irregular nesting induces extraneous work-stealing overhead).
Data-parallelism: operating on elements of a data structure in parallel

- Flat-data parallelism: operation cannot contain data-parallelism; balanced work and good match for SIMD architectures; very effective for many regular-parallel applications;

- Nested-data parallelism: operation can contain data-parallelism; well-suited to expressing irregular-parallel applications; imbalanced work and poor match for SIMD architectures

Compile nested-data parallelism to fork-join parallelism and rely on work-stealing techniques to handle load balancing (but irregular nesting induces extraneous work-stealing overhead).

Introduce *hybrid flattening*. 
Introduce *hybrid flattening*: choose what and how much to flatten

Transform *data*: represent both nested arrays and flattened arrays.

Transform *code*: introduce coercions between representations and eliminate redundant coercions.

Aggressive flattening transforms *source programs* to *flat programs*.

- source programs: no flattened arrays, no coercions
- flat programs: only flattened arrays (none of which are nested or contain pairs)

Implemented as a simple AST-to-AST transformation and optimization.
Data-Only Flattening: The Performance (mandelbrot)
Data-Only Flattening: The Performance (raytracer)

The graph shows the speedup of raytracer flattened and non-flattened with respect to the number of processors. The speedup increases linearly with the number of processors, indicating efficient scaling of the flattened raytracer compared to the non-flattened version.
Data-Only Flattening: The Performance (smvm)

The Manticore Project

FHPC'13 46 / 53
Data-Only Flattening: The Performance (dmm)

![Graph showing speedup vs. number of processors for flattened and non-flattened dense matrix multiplication.](image)
Controlled Mutable State
Controlled Mutable State

Have exploited value-oriented and mutation-free programming model:

- Free the programmer from execution details
- Provides compiler freedom to determine how to compute results
- Permits parallelism through lack of data races
- Prohibits deadlock

But, parallel programming is motivated by performance, and some problems are faster using state to share intermediate results.

Dynamic programming using a dictionary of previously computed values.

Concurrent operations having small but unpredictable overlap (limited speedup due to merging changes and Amdahl's Law).

We can write these variants in Manticore, but only as sequential programs.
Controlled Mutable State

Have exploited value-oriented and mutation-free programming model:
- Free the programmer from execution details
- Provides compiler freedom to determine how to compute results
- Permits parallelism through lack of data races
- Prohibits deadlock

But, parallel programming is motivated by performance, and some problems are faster using state to share intermediate results.
- Dynamic programming using a dictionary of previously computed values.
- Concurrent operations having small but unpredictable overlap (limited speedup due to merging changes and Amdahl’s Law).

We can write these variants in Manticore, but only as sequential programs.
Controlled Mutable State

Have exploited value-oriented and mutation-free programming model:
- Free the programmer from execution details
- Provides compiler freedom to determine how to compute results
- Permits parallelism through lack of data races
- Prohibits deadlock

But, parallel programming is motivated by performance, and some problems are faster using state to share intermediate results.
- Dynamic programming using a dictionary of previously computed values.
- Concurrent operations having small but unpredictable overlap (limited speedup due to merging changes and Amdahl’s Law).

We can write these variants in Manticore, but only as sequential programs.
0-1 Knapsack Problem:
Given a maximum weight budget and a set of items that each have a weight and value, which items should be selected subject to that weight budget to maximize the value?
0-1 Knapsack Problem:
Given a maximum weight budget and a set of items that each have a weight and value, which items should be selected subject to that weight budget to maximize the value?

```plaintext
val w = ...
val v = ...

fun knap (i, avail) = 
  if (avail = 0) orelse (i < 0) 
  then 0 
  else if Vector.sub(w, i) < avail 
  then 
    let 
      val (a, b) = 
        (knap (i-1, avail), 
        knap (i-1, avail - Vector.sub(w, i)) 
        + Vector.sub(v, i)) 
    in 
      if a > b then a else b 
    end 
  else knap (i-1, avail)
```
Memoization

0-1 Knapsack Problem:
Given a maximum weight budget and a set of items that each have a weight and value, which items should be selected subject to that weight budget to maximize the value?

```plaintext
val w = ...
val v = ...

mfun knap (i, avail) =
  if (avail = 0) orelse (i < 0)
  then 0
  else if Vector.sub(w, i) < avail
    then
      let
        val (a, b) =
          (|  knap (i-1, avail),
             knap (i-1, avail - Vector.sub(w, i))
             + Vector.sub(v, i) |)
      in
        if a > b then a else b
      end
    else knap (i-1, avail)
end
```

Fluet (RIT)
The Manticore Project
Memoization

Memoization, or *function caching*, uses a per-function lookup table to return previously computed results for identical arguments.

- First, look up the arguments in the table.
- If an entry does not exist, evaluate the function.
- Before returning the result, store it in the table.

Trade off *space for time*.
Memoization, or *function caching*, uses a per-function lookup table to return previously computed results for identical arguments.

- First, look up the arguments in the table.
- If an entry does not exist, evaluate the function.
- Before returning the result, store it in the table.

Trade off *space for time*.

Being parallelism-friendly and performant:

- Make no guarantees except functional correctness.
  - Reduced guarantees removes the need for locks and atomic operations.
- Partition the table among the NUMA nodes.
  - Balance the data across the memory banks to avoid a single bottleneck.
- Use ideas from *dynamic hash tables* to resize the table.
  - Reduce the initial memory requirement by allowing it to resize later.
  - Frequent hash conflicts result in a larger table, not worse performance.
Performance of Memoization

0-1 Knapsack Problem: 200 items / 4,000 max weight budget.

<table>
<thead>
<tr>
<th>Language</th>
<th>Sequential (s)</th>
<th>4 cores (s)</th>
<th>48 cores (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python (memoizing)</td>
<td>0.790</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td>Haskell (PLAS’12)</td>
<td>195</td>
<td>174</td>
<td>329</td>
</tr>
<tr>
<td>Manticore</td>
<td>2.24</td>
<td>1.43</td>
<td>0.574</td>
</tr>
</tbody>
</table>

How to track “too many conflicts” and decide to grow the table?

<table>
<thead>
<tr>
<th>Strategy</th>
<th>Sequential (s)</th>
<th>4 cores (s)</th>
<th>48 cores (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Element Count</td>
<td>1.85</td>
<td>1.37</td>
<td>1.51</td>
</tr>
<tr>
<td>Per-Node Element Count</td>
<td>2.15</td>
<td>1.41</td>
<td>0.877</td>
</tr>
<tr>
<td>Padded Per-Node Element Count</td>
<td>1.72</td>
<td>1.27</td>
<td>0.599</td>
</tr>
<tr>
<td>Conflict Count</td>
<td>2.22</td>
<td>1.53</td>
<td>1.50</td>
</tr>
<tr>
<td>Per-node Conflict Count</td>
<td>2.01</td>
<td>1.35</td>
<td>0.617</td>
</tr>
<tr>
<td>Padded Per-node Conflict Count</td>
<td>2.24</td>
<td>1.43</td>
<td>0.574</td>
</tr>
</tbody>
</table>
Conclusion
The Manticore Project

An effort to design and implement a parallel functional programming language aimed at general-purpose applications running on multi-core processors.

- Diverse collection of strategies for supporting parallelism
- Scalability requires designing system with parallelism in mind

manticore.cs.uchicago.edu

Questions?