Efficient and Functional Parallelism

Umut A. Acar
Department of Computer Science
Carnegie Mellon University
umut@cmu.edu
Parallel Hardware has Come of Age

• Multicore processors of all shapes and sizes
  • General Purpose Cores
  • Graphics Processing Units (GPUs)
• Per core prices anywhere from $10 to $400
8-Core AMD ($40 per core)

AMD Ryzen 7 2700X Processor with Wraith Prism LED Cooler - YD270XBGAFBOX

by AMD

⭐⭐⭐⭐⭐ 275 customer reviews | 125 answered questions

Price: $329.99 & FREE Shipping. Details

Arrives before Christmas.

Size: Processor

Processor + X470 Motherboard

Style: Ryzen 7 2700x

CPU + X470 GAMING M7 AC Ryzen 7 2700x
20-Core Intel Xeon ($160 per core)

Intel Xeon 6148 Icosa-core (20 Core) 2.40 GHz Processor - Socket 3647 Retail Pack
by Intel

Price: $3,242.82 & FREE Shipping

Get $50 off instantly: Pay $3,192.82 upon approval for the Amazon Rewards Visa Card.

Arrives before Christmas.

Service: Get professional installation Details

Without expert installation

See more
28-Core Intel Xeon  ($350 per core)

Intel Corp. Bx806738180 Xeon Pltnm 8180 Processor
by Intel

Price: $10,669.03 & FREE Shipping

Get $50 off instantly: Pay $10,619.03 upon approval for the Amazon Rewards Visa Card.

Arrives before Christmas.

Service: Get professional installation Details

Without expert installation

Include installation
+$104.47 per unit

See more
72-Core Intel: $30 per core

- Chips cost about $2500 special order from Intel

Intel® Xeon Phi™
Processors

- Host processor for highly parallel applications
- Up to 72 cores and integrated on-package memory
- Optional integrated Intel® Omni-Path Architecture fabric
4-Core Raspberry Pi ($8 per core)
Writing Parallel Programs: Lots of Choices

**Imperative languages**
- Allow effects
  \(\text{Effects} = \text{mutation/updates}\)
- Generally efficient

**Functional languages**
- Disciplined use of effects
- Generally slower

[Diagram showing various programming languages on a graph with axes for Efficiency & Performance, Imperative, and Functional. Languages include C / C++, Java, ML / OCaml, Haskell.]
Imperative languages
- Designed for sequential
- Effects = Concurrency bugs

Functional languages
- Pure functional = Parallel
- Good fit for parallel programming

Big problem with Functional
- Work Efficiency (uniprocessors)
- Scalability (multiprocessors)
- Space (memory) efficiency
This Work: “Effective” Functional Languages

- Starting point is pure functional programming
- Add support for effects
- Safe parallelism via use of effects behind an abstraction boundary.

Conjecture: Effective functional programs can be
- Work efficient
- Scalable
- Space efficient

Efficiency & Performance

Efficiency & Performance

This Work: Correctness and Efficiency

C / C++
Java
ML / OCaml
Haskell

This Work: “Effective” Functional Languages

Imperative (Buggy)
Functional (Correct)
Functional Program Example: Serial Mergesort

(* Library-level code: can use effects *)
Module Sequence =
  filter: ...
  map: ...
  partition: ...
  merge: ...

(* User-level code: purely functional *)
fun msort_serial(seq) =
  if Sequence.length(seq) <= 1 then seq
  else let (left, right) = Sequence.split_mid(seq)
      (left_sorted, right_sorted) = (msort_serial(left), msort_serial(right))
  in Sequence.merge(left_sorted, right_sorted) end
Functional Program Example: Parallel Quicksort

(* Library-level code: can use effects *)
Module Sequence =
  filter: ...
  map: ...
  partition: ...
  merge: ...

(* User-level code: purely functional *)
fun msort_par(seq) =
  if length(seq) <= 512 then msort_serial(seq)
  else let (left, right) = Sequence.split_mid seq
      (left_sorted, right_sorted) = (msort_par left || msort_par right)
    in Sequence.merge (left_sorted, right_sorted) end
What is Wrong with Functional Programming?

Many things are right but efficiency can be poor traditionally due to
• cost of abstraction
• higher order functions (polymorphism).
These are mostly solved. Many advances in the last 20 years.

The real problem: memory
• Allocate memory at much faster rates than imperative.
• Big challenge for parallel programming: memory is a shared resource.

Classic solution: per-processor heaps plus one big shared heap.
• Inefficient: Scheduling requires communication, leading to copies (promotions) to shared heap.
Our Approach

Couple tightly computation and memory.

**Goal: Exploit independencies**
- Created by functional programming, and
- Parallel Programming
Starting Point: DAG Representation

Represent a parallel program with a DAG.
• Vertices = user-level threads
• Edges = Spawn + Join

Example
• A forks B and C
• B and C join at D

B and C are independent.
(Most work in good parallel programs are independent)
Independent Heaps and Hierarchical Memory

To each thread their own memory
- Assign each thread its own heap
- Threads allocate in their own heaps

Memory hierarchy
- **Spawn**: create new heap
- **Join**: join the two heaps to that of parent.
Independent Heaps and Hierarchical Memory

To each thread their own memory
• Assign each threads its own heap
• Threads allocate in their own heaps

Memory hierarchy
• **Spawn**: create new heap
• **Join**: join the two heaps to that of parent.
Independent Heaps and Hierarchical Memory

To each thread their own memory
- Assign each thread its own heap
- Threads allocate in their own heaps

Memory hierarchy
- **Spawn**: create new heap
- **Join**: join the two heaps to that of parent.
Hierarchical Memory: Tree of Heaps

To each thread their own memory
- Assign each threads its own heap
- Threads allocate in their own heaps

Memory hierarchy
- **Spawn**: create new heap
- **Join**: join the two heaps to that of parent.
Hierarchical Memory: Disentanglement

Consider the memory graph

**Theorem:** In purely functional programs all pointers “point up” in the hierarchy [ICFP 2016].

**Proof Intuition:** a thread can only use values from its “past”. It cannot write into past objects.
Hierarchical Memory: Disentanglement

Consider the memory graph

**Theorem:** In purely functional programs all pointers “point up” in the hierarchy [ICFP 2016].

**Proof Intuition:** a thread can only use values from its “past”. It cannot write into past objects.

**Corollary:** A leaf thread can GC its heap independently of all others.
Hierarchical Memory: Disentanglement

Consider the memory graph

**Theorem:** In purely functional programs all pointers “point up” in the hierarchy [ICFP 2016].

**Proof Intuition:** a thread can only use values from its “past”. It cannot write into past objects.

**Corollary:** Concurrent heaps are independent.
Memory Management with Disentanglement

• **A hierarchy (tree) of heaps**
  • Efficient, independent allocation
  • Sharing data requires no copies

• **Garbage collection for leaf heaps:**
  • An executing thread can collects its heap.
  • No synchronization (independent)
  • Good data locality

• **Garbage collection for subtrees**
  • Like leaf heaps, but synchronize

• **Garbage collection for internal heaps:**
  • In-place collection
  • Very little synchronization needed
  • Incremental and concurrent
Theory and Practice

Problem I: fine in “theory” but inefficient in “practice”
• Creating and joining heaps at every thread operation is inefficient.

Solution I: couple heap creation/join with thread scheduler.
• When the scheduler creates a thread, create a heap.
• When two separately scheduled threads join, join their heaps.

Problem II: How to create and join heaps efficiently?
Solution II: Structure memory as a splittable/joinable data structure
# Memory: Pages, Chunks, Level Sets

## Memory in pages

<p>| | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

## Memory in chunks

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
</tr>
</thead>
</table>

## A heap

```
    5^*  A  B  C  D
        △    △    △
    6^*  E  F  G
        △    △    △
    7^*  H
        △
    8^*  I
```

## Legend

- **c**: Descriptor for chunk c
- **n^***: Level n, activated
- **n**: Level n
- **Pointer**
First Implementation and Results [PPOPP 2018]: 30% Reduction in Overheads vs Manticore

<table>
<thead>
<tr>
<th></th>
<th>mlton</th>
<th></th>
<th>manticore</th>
<th></th>
<th>mpl (our compiler)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_s$</td>
<td>$GC_s$</td>
<td>$T_1$</td>
<td>Overhead</td>
<td>$T_{72}$</td>
<td>Speed up</td>
</tr>
<tr>
<td>fib</td>
<td>2.67</td>
<td>0.0%</td>
<td>5.19</td>
<td>1.94</td>
<td>0.12</td>
<td>22.25</td>
</tr>
<tr>
<td>tabulate</td>
<td>1.11</td>
<td>39.4%</td>
<td>1.92</td>
<td>1.73</td>
<td>0.12</td>
<td>9.25</td>
</tr>
<tr>
<td>map</td>
<td>1.46</td>
<td>29.8%</td>
<td>4.02</td>
<td>2.75</td>
<td>0.49</td>
<td>2.98</td>
</tr>
<tr>
<td>reduce</td>
<td>1.13</td>
<td>40.5%</td>
<td>1.93</td>
<td>1.71</td>
<td>0.13</td>
<td>8.69</td>
</tr>
<tr>
<td>filter</td>
<td>3.62</td>
<td>11.9%</td>
<td>6.15</td>
<td>1.7</td>
<td>0.49</td>
<td>7.39</td>
</tr>
<tr>
<td>msort-pure</td>
<td>7.02</td>
<td>12.3%</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>dmm</td>
<td>3.76</td>
<td>15.2%</td>
<td>8.3</td>
<td>2.21</td>
<td>0.19</td>
<td>19.79</td>
</tr>
<tr>
<td>smvm</td>
<td>7.23</td>
<td>0.0%</td>
<td>12.68</td>
<td>1.75</td>
<td>0.32</td>
<td>22.59</td>
</tr>
<tr>
<td>strassen</td>
<td>2.54</td>
<td>1.6%</td>
<td>4.36</td>
<td>1.72</td>
<td>0.12</td>
<td>21.17</td>
</tr>
<tr>
<td>raytracer</td>
<td>7.41</td>
<td>1.3%</td>
<td>6.97</td>
<td>0.94</td>
<td>0.17</td>
<td>43.59</td>
</tr>
</tbody>
</table>
First Implementation and Results [PPOPP 2018]:
50% Increase in Speedups vs Manticore

<table>
<thead>
<tr>
<th></th>
<th>mlton</th>
<th>manticore</th>
<th>mpl (our compiler)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_s$</td>
<td>$GC_s$</td>
<td>$T_1$</td>
</tr>
<tr>
<td>fib</td>
<td>2.67</td>
<td>0.0%</td>
<td>5.19</td>
</tr>
<tr>
<td>tabulate</td>
<td>1.11</td>
<td>39.4%</td>
<td>1.92</td>
</tr>
<tr>
<td>map</td>
<td>1.46</td>
<td>29.8%</td>
<td>4.02</td>
</tr>
<tr>
<td>reduce</td>
<td>1.13</td>
<td>40.5%</td>
<td>1.93</td>
</tr>
<tr>
<td>filter</td>
<td>3.62</td>
<td>11.9%</td>
<td>6.15</td>
</tr>
<tr>
<td>msort-pure</td>
<td>7.02</td>
<td>12.3%</td>
<td>–</td>
</tr>
<tr>
<td>dmm</td>
<td>3.76</td>
<td>15.2%</td>
<td>8.3</td>
</tr>
<tr>
<td>smvm</td>
<td>7.23</td>
<td>0.0%</td>
<td>12.68</td>
</tr>
<tr>
<td>strassen</td>
<td>2.54</td>
<td>1.6%</td>
<td>4.36</td>
</tr>
<tr>
<td>raytracer</td>
<td>7.41</td>
<td>1.3%</td>
<td>6.97</td>
</tr>
</tbody>
</table>
Limitations of Purely Functional Programming

• **Church-Turing complete but**
  - Asymptotically logarithmic gap between imperative and pure algorithms
  - Constant factors are also higher, usually 2-10x

• **Can’t implement many O(1) time operations on arrays**
  - Lookup/Indexing
  - Write/Update

• **Requires more space**
  - Usually more important than time complexity

• **Caveat Emptor**: Modern functional languages usually support side effects “under the hood”. So the above does not always apply. But the model that I described thus far requires purity, complete lack of effects.
Example: Parallel Mergesort with Isolated Effects

(* Cannot use array. Could use binary trees, $O(lgn)$ extra cost *)
Module Sequence =
  filter: ...
  map: ...
  partition: ...
  merge: ...

(* Cannot use imperative in-place quicksort at the base case *)
fun msort_par seq =
  if length seq <= 512 then qsort_serial(seq)
  else let (left, right) = Sequence.split_mid seq
        (left_sorted, right_sorted) = (msort_par left || msort_par right)
    in Sequence.merge (left_sorted, right_sorted) end
Example: Parallel Array Map

(* Cannot be implemented efficiently, requires writing into an array *)

fun map f arr =
  let n = length(arr)
    result = new array[n]
  in parfor i = 0 to n do
    result[i] := f(arr[i])
  end
Extending Disentanglement to Allow Effects

Allow arbitrary writes/reads from mutable data

A computation is *disentangled* if for any read of an object $x$ by some thread $t$ the thread that allocates $x$ precedes $t$ in the computation DAG.

**Intuition:** A thread can only read data that is created by an ancestor thread. It gains no new *knowledge* from other threads.

- **No cross pointers:** Disentanglement disallows communication between concurrent threads
- Disentanglement is not race freedom.
How Common is Disentanglement?

- **Many parallel algorithms are disentangled**
  - **Array operations**: create, initialize, map, filter, reduce, scan, update, inject.
  - **Sorting**: mergesort, quicksort, sample sort
  - **Statistics**: deduplication, histogramming.
  - **Matrix calculations**: dense/sparse matrix multiply…
  - **Tree algorithms**: Barnes-Hut, nearest neighbors
  - **Graph algorithms**: BFS, DFS, graph contraction, MSTs…

- **Why?** The reason is that entanglement usually means data races!

- **Conjecture**: Determinacy-race-free parallel programs are disentangled.

- Purely functional composition of “observationally pure” expressions are disentangled. (Can come back to this if we have time.)
What is Disentanglement Good for?

Disentanglement is not a topological property of computation graphs.

But it implies one: consider memory graph partitioned across heaps in the hierarchy:

- An edge can “point up”
- An edge can “point down”
- There are no cross edges
What is Disentanglement Good for?

Disentanglement is not a topological property of computation graphs.

But it implies one: consider memory graph partitioned across heaps in the hierarchy:
- An edge can “point up”
- An edge can “point down”
- There are no cross edges

Concurrent heaps are still independent!
Memory Management with Disentangled Effects

Similar to pure hierarchy
- Assign each thread its own heap
  - Create heaps when spawning
  - Join heaps when joining threads

But there are differences
- Track “down pointers”
- Down pointers are roots in a GC
- Eagerly promote (copy) down pointers to ancestor heaps during GC.

Glossing over many details [WYAF’18]
Implementation in MPL

• Extended the Standard ML to support parallelism and effects [WYAF’18]
• Current implementation performs leaf collections only
• Implemented a variety of disentangled benchmarks: pure and effectful
  • Effects are usually confined to data structures.
  • Algorithms are pure.
• Empirical evaluation on a 72 Core Machine
• At Carnegie Mellon, we teach Algorithms in ML.
  • We started using MPL in this class.
MPL with Disentangled Effects

- **Small overheads:** ~20%
- **Good Speedups:** ~40 with 72 cores

<table>
<thead>
<tr>
<th></th>
<th>MLton $T_s$</th>
<th>MLton $T_1$</th>
<th>Overhead</th>
<th>mpl (Ours) $T_{72}$</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib</td>
<td>3.9</td>
<td>5.0</td>
<td>1.3</td>
<td>.085</td>
<td>45.9</td>
</tr>
<tr>
<td>tabulate</td>
<td>2.5</td>
<td>2.5</td>
<td>1.1</td>
<td>.059</td>
<td>39.0</td>
</tr>
<tr>
<td>map</td>
<td>3.7</td>
<td>2.9</td>
<td>1.1</td>
<td>.066</td>
<td>39.4</td>
</tr>
<tr>
<td>map-in-place</td>
<td>3.7</td>
<td>3.0</td>
<td>1.2</td>
<td>.065</td>
<td>38.5</td>
</tr>
<tr>
<td>scan</td>
<td>5.6</td>
<td>4.6</td>
<td>1.1</td>
<td>.11</td>
<td>39.1</td>
</tr>
<tr>
<td>reduce</td>
<td>2.9</td>
<td>2.2</td>
<td>1.0</td>
<td>.039</td>
<td>53.8</td>
</tr>
<tr>
<td>filter</td>
<td>5.0</td>
<td>3.9</td>
<td>1.1</td>
<td>.075</td>
<td>49.3</td>
</tr>
<tr>
<td>samplesort</td>
<td>12.3</td>
<td>13.3</td>
<td>1.1</td>
<td>.34</td>
<td>36.2</td>
</tr>
<tr>
<td>mergesort</td>
<td>13.9</td>
<td>12.5</td>
<td>.93</td>
<td>.4</td>
<td>33.5</td>
</tr>
<tr>
<td>dmm</td>
<td>13.6</td>
<td>23.9</td>
<td>1.8</td>
<td>.45</td>
<td>30.2</td>
</tr>
<tr>
<td>dedup</td>
<td>6.7</td>
<td>3.4</td>
<td>.97</td>
<td>.1</td>
<td>35.0</td>
</tr>
<tr>
<td>histogram</td>
<td>6.5</td>
<td>3.9</td>
<td>1.1</td>
<td>.13</td>
<td>26.9</td>
</tr>
<tr>
<td>barnes-hut</td>
<td>9.5</td>
<td>8.2</td>
<td>1.1</td>
<td>.2</td>
<td>37.0</td>
</tr>
<tr>
<td>all-nearest</td>
<td>4.8</td>
<td>3.1</td>
<td>1.0</td>
<td>.11</td>
<td>28.2</td>
</tr>
</tbody>
</table>
Scalability (Speedup)
Space Usage: Usually Within 2X of Sequential

- **Surprising:** would expect higher space consumption

- **Reason:** leaf collections are efficient and aggressive in reclaiming memory.

<table>
<thead>
<tr>
<th></th>
<th>Working Size</th>
<th>mlton</th>
<th>mpl (Ours)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$R_S$</td>
<td>$R_1$</td>
<td>$R_{72}$</td>
</tr>
<tr>
<td>fib</td>
<td>20.0 K</td>
<td>1.9 M</td>
<td>3.2 M</td>
</tr>
<tr>
<td>tabulate</td>
<td>538.7 M</td>
<td>548.6 M</td>
<td>1.5 G</td>
</tr>
<tr>
<td>map</td>
<td>1.1 G</td>
<td>1.3 G</td>
<td>1.6 G</td>
</tr>
<tr>
<td>map-in-place</td>
<td>559.5 M</td>
<td>1.2 G</td>
<td>1.5 G</td>
</tr>
<tr>
<td>scan</td>
<td>1.0 G</td>
<td>1.5 G</td>
<td>1.9 G</td>
</tr>
<tr>
<td>reduce</td>
<td>559.5 M</td>
<td>1.4 G</td>
<td>1.6 G</td>
</tr>
<tr>
<td>filter</td>
<td>569.5 M</td>
<td>1.2 G</td>
<td>1.6 G</td>
</tr>
<tr>
<td>samplesort</td>
<td>679.5 M</td>
<td>1.1 G</td>
<td>1.8 G</td>
</tr>
<tr>
<td>mergesort</td>
<td>639.5 M</td>
<td>1.5 G</td>
<td>2.6 G</td>
</tr>
<tr>
<td>dmm</td>
<td>99.8 M</td>
<td>134.0 M</td>
<td>174.4 M</td>
</tr>
<tr>
<td>dedup</td>
<td>659.4 M</td>
<td>1.3 G</td>
<td>1.9 G</td>
</tr>
<tr>
<td>histogram</td>
<td>828.8 M</td>
<td>1.5 G</td>
<td>2.0 G</td>
</tr>
<tr>
<td>barnes-hut</td>
<td>97.4 M</td>
<td>244.7 M</td>
<td>373.9 M</td>
</tr>
<tr>
<td>all-nearest</td>
<td>98.4 M</td>
<td>504.0 M</td>
<td>664.6 M</td>
</tr>
</tbody>
</table>
### Comparison to Java: Usually Scales Better

<table>
<thead>
<tr>
<th></th>
<th>1 Core Time</th>
<th>72 Core Time</th>
<th>1 Core Space</th>
<th>72 Core Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>Java Ours</td>
<td>$T_{72}$</td>
<td>Java Ours</td>
</tr>
<tr>
<td>map</td>
<td>3.20</td>
<td>1.10</td>
<td>1.15</td>
<td>17.39</td>
</tr>
<tr>
<td>tabulate</td>
<td>2.21</td>
<td>0.88</td>
<td>0.93</td>
<td>15.68</td>
</tr>
<tr>
<td>scan</td>
<td>6.19</td>
<td>1.35</td>
<td>1.14</td>
<td>10.34</td>
</tr>
<tr>
<td>histogram</td>
<td>16.59</td>
<td>4.25</td>
<td>1.10</td>
<td>8.45</td>
</tr>
<tr>
<td>dedup</td>
<td>10.82</td>
<td>3.18</td>
<td>0.72</td>
<td>7.25</td>
</tr>
<tr>
<td>map-in-place</td>
<td>2.55</td>
<td>0.85</td>
<td>0.41</td>
<td>6.23</td>
</tr>
<tr>
<td>sort$^\dagger$</td>
<td>12.69</td>
<td>0.95</td>
<td>0.90</td>
<td>2.64</td>
</tr>
<tr>
<td>reduce</td>
<td>1.70</td>
<td>0.77</td>
<td>0.07</td>
<td>1.90</td>
</tr>
<tr>
<td>filter</td>
<td>1.27</td>
<td>0.33</td>
<td>0.08</td>
<td>1.12</td>
</tr>
<tr>
<td>fib</td>
<td>1.36</td>
<td>0.27</td>
<td>0.04</td>
<td>0.51</td>
</tr>
<tr>
<td>dmm</td>
<td>5.70</td>
<td>0.24</td>
<td>0.17</td>
<td>0.39</td>
</tr>
<tr>
<td>Mean</td>
<td>-</td>
<td>0.88</td>
<td>-</td>
<td>3.57</td>
</tr>
</tbody>
</table>

$^\dagger$ Compared to our samplesort.
Comparison to Java: Uses Less Space

<table>
<thead>
<tr>
<th></th>
<th>1 Core Time</th>
<th>72 Core Time</th>
<th>1 Core Space</th>
<th>72 Core Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$ Java</td>
<td>Ours $T_{72}$</td>
<td>Java Ours</td>
<td>$R_1$ Java Ours</td>
</tr>
<tr>
<td>map</td>
<td>3.20</td>
<td>1.10</td>
<td>1.15</td>
<td>17.39</td>
</tr>
<tr>
<td>tabulate</td>
<td>2.21</td>
<td>0.88</td>
<td>0.93</td>
<td>15.68</td>
</tr>
<tr>
<td>scan</td>
<td>6.19</td>
<td>1.35</td>
<td>1.14</td>
<td>10.34</td>
</tr>
<tr>
<td>histogram</td>
<td>16.59</td>
<td>4.25</td>
<td>1.10</td>
<td>8.45</td>
</tr>
<tr>
<td>dedup</td>
<td>10.82</td>
<td>3.18</td>
<td>0.72</td>
<td>7.25</td>
</tr>
<tr>
<td>map-in-place</td>
<td>2.55</td>
<td>0.85</td>
<td>0.41</td>
<td>6.23</td>
</tr>
<tr>
<td>sort†</td>
<td>12.69</td>
<td>0.95</td>
<td>0.90</td>
<td>2.64</td>
</tr>
<tr>
<td>reduce</td>
<td>1.70</td>
<td>0.77</td>
<td>0.07</td>
<td>1.90</td>
</tr>
<tr>
<td>filter</td>
<td>1.27</td>
<td>0.33</td>
<td>0.08</td>
<td>1.12</td>
</tr>
<tr>
<td>fib</td>
<td>1.36</td>
<td>0.27</td>
<td>0.04</td>
<td>0.51</td>
</tr>
<tr>
<td>dmm</td>
<td>5.70</td>
<td>0.24</td>
<td>0.17</td>
<td>0.39</td>
</tr>
<tr>
<td>Mean</td>
<td>-</td>
<td>0.88</td>
<td>-</td>
<td>3.57</td>
</tr>
</tbody>
</table>

† Compared to our samplesort.
Impact of Purity: 3X in Work, Space, and Time

<table>
<thead>
<tr>
<th></th>
<th>1 Core Time</th>
<th>72 Core Time</th>
<th>1 Core Space</th>
<th>72 Core Space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$T_1$</td>
<td>$T_{72}$</td>
<td>$R_1$</td>
<td>$R_{72}$</td>
</tr>
<tr>
<td></td>
<td>Manticore</td>
<td>Manticore</td>
<td>Manticore</td>
<td>Manticore</td>
</tr>
<tr>
<td></td>
<td>Ours</td>
<td>Ours</td>
<td>Ours</td>
<td>Ours</td>
</tr>
<tr>
<td>fib</td>
<td>7.9</td>
<td>0.13</td>
<td>7.0 M</td>
<td>466.7 M</td>
</tr>
<tr>
<td></td>
<td>1.6</td>
<td>1.6</td>
<td>2.2</td>
<td>13.9</td>
</tr>
<tr>
<td>tabulate</td>
<td>6.5</td>
<td>0.16</td>
<td>1.3 G</td>
<td>2.7 G</td>
</tr>
<tr>
<td></td>
<td>2.6</td>
<td>2.8</td>
<td>0.84</td>
<td>3.4</td>
</tr>
<tr>
<td>map</td>
<td>9.9</td>
<td>0.22</td>
<td>4.2 G</td>
<td>2.9 G</td>
</tr>
<tr>
<td></td>
<td>3.4</td>
<td>3.3</td>
<td>2.7</td>
<td>2.0</td>
</tr>
<tr>
<td>filter</td>
<td>11.5</td>
<td>0.2</td>
<td>2.6 G</td>
<td>2.7 G</td>
</tr>
<tr>
<td></td>
<td>2.9</td>
<td>2.7</td>
<td>1.7</td>
<td>2.7</td>
</tr>
<tr>
<td>reduce</td>
<td>7.6</td>
<td>0.14</td>
<td>1.3 G</td>
<td>2.8 G</td>
</tr>
<tr>
<td></td>
<td>3.5</td>
<td>3.6</td>
<td>0.82</td>
<td>2.0</td>
</tr>
</tbody>
</table>
This Work: “Effective” Functional Languages

- Starting point is pure functional programming
- Add support for effects
- Safe parallelism via use of effects behind an abstraction boundary.

Conjecture: Effective functional programs can be
- Work efficient
- Scalable
- Space efficient

Efficiency & Performance

C / C++
Java
ML / OCaml
Haskell

This Work

Imperative
Functional
Summary and Conclusion

Ignoring efficiency, functional programming is great for parallelism
  • Allows for disciplined use of effects
  • Supports (and invented) higher order functions (e.g., map, reduce).

This talk: Efficiency is doable
Insight: use the independencies created by functional & parallel programs.
Results show significant improvements in the state of the art.
Careful use of effects is crucial for efficiency.

Conjecture: parallel functional programming can be the most efficient of all (except C).
Thank You! Questions?
Observational Purity and Disentanglement

• Recall: A computation is *disentangled* if for any read of an object \(x\) by some thread \(t\) the thread that allocates \(x\) precedes \(t\) in the computation DAG.

• We say that a computation is *observationally pure*, if the writes all target objects allocated by threads in the DAG.

• **Intuition**: Observationally pure programs do not have observable effects. They use effects purely to improve efficiency and performance.

• **Proposition**: Purely functional composition of all disentangled and observationally pure computations are disentangled and observationally pure.