Functional Parallel Algorithms

Guy Blelloch
Carnegie Mellon University
Parallelism is here... And Growing!
Some benchmarks

Speedups on 32 cores (Dell Poweredge):

- **Comparison sorting**: 24x speedup
  - Sample sort (1 billion strings in 12 secs.)
- **Minimum Spanning Tree**: 17x speedup
  - Parallel Kruskal (1 billion edges in 8 secs.)
- **K-nearest Neighbors**: 14x speedup
  - Oct-tree (.1 billion points in 30 secs.)
- **Delaunay Triangulation**: 20x speedup
  - Incremental (.1 billion points in 48 secs.)
- **Dictionary Insert+Lookup**: 27x speedup
  - Hashing (1 billion strings in 6 secs.)
The State of Parallel Algorithms

• No accepted model by the algorithms/complexity community.
• 136 papers Accepted to 2011 ACM/SIAM Symposium on Discrete Algorithms (SODA). 0 of them are about parallel algorithms.
Opportunity for the PL Community

Reasons PL community can play a major role in how people will program and analyze parallel algorithms.

– Understand how to control effects
– Errors matter now
– Ease of programming matters
– Language based cost models
– “Parallel Thinking” is more natural.
Parallelism vs. Concurrency

- **Parallelism**: using multiple processors/cores running at the same time. Property of the machine.
- **Concurrency**: non-determinacy due to interleaving threads. Property of the application.

<table>
<thead>
<tr>
<th>Parallelism</th>
<th>Concurrency</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>sequential</td>
</tr>
<tr>
<td>parallel</td>
<td>concurrent</td>
</tr>
<tr>
<td>serial</td>
<td>Traditional programming</td>
</tr>
<tr>
<td>parallel</td>
<td>Deterministic parallelism</td>
</tr>
<tr>
<td></td>
<td>General parallelism</td>
</tr>
</tbody>
</table>
procedure QUICKSORT(S):
    if S contains at most one element then return S
    else
        begin
            choose an element a randomly from S;
            let S₁, S₂ and S₃ be the sequences of elements in S less than, equal to, and greater than a, respectively;
            return (QUICKSORT(S₁) followed by S₂ followed by QUICKSORT(S₃))
        end
But....

We need a way to compare algorithms.
  – How “parallel” is quicksort
  – How does it compare to other sorting algorithms

We need a **formal cost model** so that we can make concrete claims.
Language Based Cost Models

A cost model based on the operational semantics +
Provable implementation bounds
Call-by-value $\lambda$-calculus

$\lambda x. e \Downarrow \lambda x. e$ \hspace{1cm} \text{(LAM)}

\[
\begin{array}{c}
\frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v \quad e[v/x] \Downarrow v'}{e_1 e_2 \Downarrow v'} \quad \text{(APP)}
\end{array}
\]
The Parallel $\lambda$-calculus: cost model

$$e \Downarrow v; w,d$$

Reads: expression $e$ evaluates to $v$ with work $w$ and span $d$.

- **Work** (W): sequential work
- **Span** (D): parallel depth
The Parallel $\lambda$-calculus: cost model

\[
\lambda x. e \Downarrow \lambda x. e; 1,1 \quad \text{(LAM)}
\]

\[
e_1 \Downarrow \lambda x. e; w_1, d_1 \quad e_2 \Downarrow v; w_2, d_2 \quad e[v/x] \Downarrow v'; w_3, d_3
\]

\[
e_1 e_2 \Downarrow v'; 1 + w_1 + w_2 + w_3, 1 + \max(d_1,d_2) + d_3 \quad \text{(APP)}
\]

Work adds
Span adds sequentially, and max in parallel
The Parallel $\lambda$-calculus: cost model

\[
\text{let, letrec, datatypes, tuples, case-statement can all be implemented with constant overhead}
\]

Integers and integer operations ($+, <, \ldots$) can be implemented with $O(\log n)$ cost for integers up to $n$
The Parallel $\lambda$-calculus (constants)

\[
\begin{align*}
\text{(CONST)} & \quad c \Downarrow c; \begin{array}{|c|c|}
\hline
1 & 1 \\
\hline
\end{array} \\
\text{(APPC)} & \quad e_1 \Downarrow c; \begin{array}{|c|c|}
\hline
w_1 & d_1 \\
\hline
\end{array} \quad e_2 \Downarrow v; \begin{array}{|c|c|}
\hline
w_2 & d_2 \\
\hline
\end{array} \quad \delta(c,v) \Downarrow v' \\
\quad e_1 e_2 \Downarrow v'; \begin{array}{|c|c|}
\hline
1 + w_1 + w_2 & 1 + \max(d_1,d_2) \\
\hline
\end{array} \\
\end{align*}
\]

\[c_n = 0, \ldots, n, +, +_0, \ldots, +_n, <, <_0, \ldots, <_n, ×, ×_0, \ldots, ×_n, \ldots \quad \text{(constants)}\]
The Parallel $\lambda$-calculus cost model

$\lambda x. e \Downarrow \lambda x. e; 1,1$  \hspace{1cm} (LAM)

$\frac{e_1 \Downarrow \lambda x. e; w_1,d_1 \quad e_2 \Downarrow v; w_2,d_2 \quad e[v/x] \Downarrow v'; w_3,d_3}{e_1 e_2 \Downarrow v'; 1 + w_1 + w_2 + w_3, 1 + \max(d_1,d_2) + d_3}$  \hspace{1cm} (APP)

$c \Downarrow c; 1,1$  \hspace{1cm} (CONST)

$\frac{e_1 \Downarrow c; w_1,d_1 \quad e_2 \Downarrow v; w_2,d_2 \quad \delta(c,v) \Downarrow v'}{e_1 e_2 \Downarrow v'; 1 + w_1 + w_2, 1 + \max(d_1,d_2)}$  \hspace{1cm} (APPC)

$c_n = 0, \cdots, n, +, +_0, \cdots, +_n, <, <_0, \cdots, <_n, \times, \times_0, \cdots, \times_n, \cdots$  \hspace{1cm} (constants)
The Second Half: Provable Implementation Bounds

Theorem [FPCA95]: If $e \Downarrow v; w,d$ then $v$ can be calculated from $e$ on a CREW PRAM with $p$ processors in $O\left(\frac{w}{p} + d \log p\right)$ time.

Can’t really do better than: $\max\left(\frac{w}{p},d\right)$

If $w/p > d \log p$ then “work dominates”

We refer to $w/p$ as the parallelism.
procedure QUICKSORT(S):
  if S contains at most one element then return S
  else
    begin
      choose an element a randomly from S;
      let $S_1$, $S_2$ and $S_3$ be the sequences of elements in $S$ less than, equal to, and greater than a, respectively;
      return (QUICKSORT($S_1$) followed by $S_2$ followed by QUICKSORT($S_3$))
    end
fun qsort [] = []
  | qsort S =
    let val a::_ = S
      val S_1 = filter (fn x => x < a) S
      val S_2 = filter (fn x => x = a) S
      val S_3 = filter (fn x => x > a) S
    in
      append (qsort S_1) (append S_2 (qsort S_3))
    end
Qsort Complexity

Sequential Partition
Parallel calls

Partition
(less than, ...)

Span = O(n)

All bounds expected case over all inputs of size n

Work = O(n \log n)

Parallelism = O(\log n)

Not a very good parallel algorithm
Tree Quicksort

datatype 'a seq = Empty
    | Leaf of 'a
    | Node of 'a seq * 'a seq

fun append Empty b = b
    | append a Empty = a
    | append a b = Node(a,b)

fun filter f Empty = Empty
    | filter f (Leaf x) =
        if (f x) the Leaf x else Empty
    | filter f Node(l,r) =
        append (filter f l) (filter f r)
Tree Quicksort

fun qsort Empty = Empty
  | qsort S =
    let val a = first S
      val S_1 = filter (fn x => x < a) S
      val S_2 = filter (fn x => x = a) S
      val S_3 = filter (fn x => x > a) S
    in
      append (qsort S_1) (append S_2 (qsort S_3))
    end
Qsort Complexity

Parallel partition
Parallel calls

Span = $O(lg \ n)$

Work = $O(n \ log \ n)$

Span = $O(lg^2 \ n)$

A good parallel algorithm

Parallelism = $O(n/log \ n)$
Example: Merging

\[
\begin{align*}
\text{Merge} & \quad ([], l2) = l2 \\
& \mid (l1, []) = l1 \\
& \mid (h1::t1, h2::t2) = \\
& \quad \text{if } (h1 < h2) \quad h1::\text{Merge}(t1, h2::t2) \\
& \quad \text{else } h2::\text{Merge}(h1::t1, t2)
\end{align*}
\]
The Split Operation

datatype 'a seq = Empty
  | Node of 'a * 'a seq * 'a seq

fun split (p, Empty) = (Empty, Empty)
  | split (p, node(v, L, R)) =
    if p < v then
      let val (L1 ,R1) = split(p ,L)
      in (L1,node(v, R1, R)) end
    else
      let val (L1,R1) = split(p ,R)
      in (node (v, L, Ll), R1) end;
Merging

\[ \text{Merge}(A,B) = \]
\[
\text{let}
\]
\[
\text{Node}(A_L, m, A_R) = A
\]
\[
(B_L, B_R) = \text{split}(B, m)
\]
\[
in
\]
\[
\text{Node}(\text{Merge}(A_L, B_L), m, \text{Merge}(A_R, B_R))
\]

Span = \(O(\log^2 n)\)
Work = \(O(n)\)

Merge in parallel
Adding Functional Arrays: NESL

\[ \{e_1 : x \text{ in } e_2 \mid e_3\} \]

\[
e'[v_i/x] \downarrow v_i';\; w_i, d_i \quad i \in \{1\ldots n\}
\]

\[
\{e' : x \text{ in } [v_1\ldots v_n]\} \downarrow [v_1'\ldots v_n'];\; 1 + \sum_{i=1}^{n} w_i, 1 + \max_{i=1}^{v} d_i
\]

Primitives:

\[- : \text{`a seq * (int,`a) seq -> `a seq}\]

• \([g,c,a,p] \leftarrow [(0,d),(2,f),(0,i)]\]
  \[i,c,f,p]\]

elt, index, length  

[ICFP95]
QuickSort in NESL

function quicksort(S) =  
if (#S <= 1) then S  
else let  
    a = S[elt(#S)];  
    S1 = {e in S | e < a};  
    S2 = {e in S | e = a};  
    S3 = {e in S | e > a};  
    R = {quicksort(v) : v in [S1, S3]};  
in R[0] ++ S2 ++ R[1];  

Span = O(log n)  
Work = O(n)  
Space = O(n)  
Expected
Provable Implementation Bounds

Theorem: If $e \downarrow v; w,d,s$ then $v$ can be calculated from $e$ on a CREW PRAM with $p$ processors in $O\left(\frac{w}{p} + d \log p\right)$ time and $O(s + pd \log p)$ space.
Interesting Side Note

Can implement hash tables so insertion of n elements takes:

\[ W(n) = O(n) \text{ and } D(n) = O(\log n) \] expected case

Search takes \( D(n) = W(n) = O(1) \) expected case
Example: Graph Connectivity

Form stars

contract

relabel
Example : Graph Connectivity

Edge List Representation:

Edges = [(0,1), (0,2), (2,3), (3,4), (3,5), (3,6), (1,3), (1,5), (5,6), (4,6)]

Hooks = [(0,1), (1,3), (1,5), (3,6), (4,6)]
Example: Graph Connectivity

L = Vertex Labels, E = Edge List

function connectivity(L, E) =
if #E = 0 then L
else let
  FL = \{coinToss(.5) : x in [0:#L]\};
  H = \{(u,v) in E | FL[u] and not(FL[v])\};
  L = L <- H;
  E = \{(L[u],L[v]) : (u,v) in E | L[u] \neq L[v]\};
in connectivity(L,E);

D = O(log n)
W = O(m log n)
Some Unfinished Problems

• How to take account of locality in a high-level way.
• Dealing properly with randomness
• Dealing properly with exceptions
• Efficient purely functional algorithms for many problems.
Summary

• Purely functional algorithms have several more advantages in parallel than sequentially.
• Programming-based cost models and implementation bounds could change the way people think about costs and open the door to all sorts of other “abstract” costs.
• Functional parallel algorithms are fun!!!!