Parallel Thinking*

Guy Blelloch

Carnegie Mellon University

*PROBE as part of the Center for Computational Thinking
CPU-Frequency 1993 - 2005
AMD and Intel

Year

Frequency [MHz]

0  500  1000  1500  2000  2500  3000  3500  4000


AMD
Intel

via Michael Scott
Parallelism is here... And Growing!

Number of Cores

2006  2007  2008  2009  2010  ...  2015

Parallelism for the Masses
"Opportunities and Challenges"

Andrew Chien, 2008
Parallel Thinking

How to deal with parallelism/ concurrency:

Option I: Minimize what users have to learn about parallelism. Hide parallelism in libraries which are programmed by a few experts.

Option II: Teach parallelism as an advanced subject after and based on standard material on sequential computing.

Option III: Teach parallelism from the start with sequential computing as a special case.
Parallel Thinking

Could it be that it is more natural to think about parallel algorithms than sequential algorithms?

If done right could parallel programming be easier than sequential programming, or at least for most uses?

Are we currently brainwashing students to think sequentially?

What are the core parallel ideas that all computer scientists should know?
Quicksort from Sedgewick

public void quickSort(int[] a, int left, int right) {
    int i = left-1;  int j = right;
    if (right <= left) return;
    while (true) {
        while (a[++i] < a[right]);
        while (a[right]<a[--j])
            if (j==left) break;
        if (i >= j) break;
        swap(a,i,j); }
    swap(a, i, right);
    quickSort(a, left, i - 1);
    quickSort(a, i+1, right); }

Sequential!
Quicksort from Aho-Hopcroft-Ullman

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

Two forms of natural parallelism
Lesson 1

Natural parallelism is often lost in “low-level” implementations.

- We need “higher level” languages
- We need to revert back to the core ideas of an algorithm
Quicksort in NESL

function quicksort(S) =
if (#S <= 1) then S
else let
    a = S[rand(#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];
Parallel selection

\{ e \in S \mid e < a \};

\begin{align*}
S &= [2, 1, 4, 0, 3, 1, 5, 7] \\
F = S < 4 &= [1, 1, 0, 1, 1, 1, 0, 0] \\
I = \text{addscan}(F) &= [0, 1, 2, 2, 3, 4, 5, 5] \\
\text{where } F \\
R[I] = S &= [2, 1, 0, 3, 1]
\end{align*}

Each element gets sum of previous elements. Seems sequential?
Scan

\[
\begin{align*}
\text{sum} & \quad [2, 1, 4, 2, 3, 1, 5, 7] \\
\text{recurse} & \quad [3, 6, 4, 12] \\
\text{sum} & \quad [0, 3, 9, 13] \\
\text{interleave} & \quad [2, 7, 12, 18] \\
\text{sum} & \quad [0, 2, 3, 7, 9, 12, 13, 18]
\end{align*}
\]
Scan code

function scan(A, op) =
if (#A <= 1) then [0]
else let
  sums = \{op(A[2*i], A[2*i+1]) : i in [0:#a/2]\};
  evens = scan(sums, op);
  odds = \{op(evens[i], A[2*i]) : i in [0:#a/2]\};
in interleave(evens, odds);

A = [2, 1, 4, 2, 3, 1, 5, 7]
sums = [3, 6, 4, 12]
evens = [0, 3, 9, 13] (result of recursion)
odd = [2, 7, 12, 18]
result = [0, 2, 3, 7, 9, 12, 13, 18]
Lessons 2 and 3

Just because it seems sequential does not mean it is

And

When in doubt recurse on a smaller problem
QuickSort in Cilk++

```cilk
seq quicksort(seq S) {
    if (S.length < 2) return S;
    double a = S[rand(S.length)];
    seq S1, S2, S3;
    cilk_spawn S1 = quicksort(lessThan(S, a));
    cilk_spawn S2 = eqTo(S, a);
    S3 = quicksort(grThan(S, a));
    cilk_sync;
    return S1.append(S2.append(S3));
}
```
Quicksort in Cilk++

```c
int cnt=0;

seq quicksort(seq S) {
    cnt++; // ??
    if (S.length < 2) return S;
    double a = S[rand(S.length)];
    seq S1,S2,S3;
    cilk_spawn S1 = quicksort(lessThan(S,a));
    cilk_spawn S2 = eqTo(S,a);
    S3 = quicksort(grThan(S,a));
    cilk_sync;
    S1.append(S2.append(S3));
}
```
Lesson 4

Deterministic parallelism is important
- Functional languages
- Race detectors
- Using non-functional languages in a functional style

Atomic regions and transactions don’t solve this problem.
Qsort Complexity

Sequential Partition
Parallel calls

Span = O(n)
Work = O(n log n)

Not a very good parallel algorithm

partition
(less than, …)
append
subroutine quicksort(a,n)
integer n,nless,less(n),greater(n),a(n)
if (n < 2) return
pivot = a(1)
nless = count(a < pivot)
less = pack(a, a < pivot)
greater = pack(a, a >= pivot)
call quicksort(less, nless)
a(1:nless) = less
call quicksort(greater, n-nless)
a(nless+1:n) = less
end subroutine
Qsort Complexity

Parallel partition
Sequential calls

Span = \( O(n) \)

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

Still not a very good parallel algorithm
Qsort Complexity

Parallel partition
Parallel calls

Work = $O(n \log n)$

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

A good parallel algorithm
Combining for parallel map:

\[
p_{\text{exp}} = \{ \exp(e) : e \text{ in } A \}
\]

\[
W_{p_{\text{exp}}}(A) = \sum_{i=0}^{n-1} W_{\exp}(A_i)
\]

\[
D_{p_{\text{exp}}}(A) = \max_{i=0}^{n-1} D_{\exp}(A_i)
\]

Can prove runtime bounds for various models:

e.g. \( T = O(W/P + D \log P) \)
Generally for a DAG

Any “greedy” schedule for a DAG with span (depth) $D$ and work (size) $W$ will complete in:

$$T < \frac{W}{P} + D$$

time

But, different schedules have very different space characteristics
Lessons 5, 6, 7 and 8

- Abstract cost models that are not machine based are important. Perhaps based in semantics of language.

- Work and span are reasonable measures.

- Need to take advantage of both “data parallelism” and “function parallelism”.

- Scheduling is important.
Quicksort in Multilisp

(defun quicksort (L) (qs L nil))

(defun qs (L rest)
  (if (null L) rest
    (let ((a (car L))
              (L1 (filter (lambda (b) (< b a)) (cdr L)))
              (L3 (filter (lambda (b) (>= b a)) (cdr L))))
      (qs L1 (future (cons a (qs L3 rest)))))))

(defun filter (f L)
  (if (null L) nil
    (if (f (car L))
        (future (cons (car L) (filter f (cdr L)))
        (filter f (cdr L))))))
Quicksort in Multilisp (futures)

Span = $O(n)$

Work = $O(n \log n)$

Not a very good parallel algorithm
Lesson 9

Pipelining might be useful but one should analyze span before using it blindly.
Example: Merging

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

What about in parallel?
Merging

Merge(A,B) =
   let
      Node(A_L, m, A_R) = A
      (B_L ,B_R) = split(B, m)
   in
      Node(Merge(A_L,B_L), m, Merge(A_R,B_R))

Span = O(log² n)
Work = O(n)

Merge in parallel
Merging with Futures

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

\[\text{Span} = O(\log n)\]
\[\text{Work} = O(n)\]
Lessons 10, 11, and 9.5

- Divide and conquer even more useful in parallel than sequentially

and

- Trees are better than lists for parallelism

and

- (9.5) Pipelining is useful
Example: Graph Connectivity

Form stars

relabel

contract
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

$L = \text{Vertex Labels}, \quad H = \text{hooks} \quad E = \text{Edge List}$

$L = L \leftarrow H; \quad /* \text{Write tails to heads} */$

$E = \{(L[u],L[v]) \quad /* \text{update edge labels} */$
$\quad : (u,v) \text{ in } E$
$\quad \mid L[u] \neq L[v]\}; \quad /* \text{remove self edges} */$
Example: Graph Connectivity

How do we find the stars?

The world looks the same from every node. We need to break symmetry.
Example: Random Mate

$L = \text{Vertex Labels, } E = \text{Edge List}$

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

$D = O(\log n)$
$W = O(m \log n)$
Lessons 3, 12, 13

- When in doubt recurse on a smaller problem
- and
- We often need to break symmetry
- and
- Sampling is useful
The Lessons

General:
1. Natural parallelism is often lost in “low-level” implementations.
2. Just because it seems sequential does not mean it is

Model and Language:
4. Deterministic parallelism is important
5. Abstract cost models that are not machine based are important.
6. Work and span are reasonable measures
7. Need to take advantage of both “data” and “function” parallelism
8. Scheduling is important

Algorithmic Techniques
3. When in doubt recurse on a smaller problem
9. Pipelining is useful, with care
10. Divide and conquer even more useful in parallel
11. Trees are better than lists for parallelism
12. We often need to break symmetry
13. Sampling is useful
What else

Non-deterministic parallelism:
- Races and race detection
- Sequential consistency, serializability, linearizability, atomic primitives, locking techniques, transactions
- Concurrency models, e.g. the pi-calculus
- Lock and wait free algorithms

Architectural issues
- Cache coherence, memory layout, latency hiding
- Network topology, latency vs. throughput
- ...

...
Acknowledgements

This talk has been based on 30 years of research on parallelism by 100s of people.
Many ideas from the PRAM (theory) community and PL community
Conclusions/Questions

Should we teach parallelism from day 1 and sequential computing as a special case?

Could teaching parallelism actually make some things easier?

Are there a reasonably small number of core ideas that every undergraduate needs to know? If so, what are they?