Should we Teach Parallelism Throughout our Data Structures and Algorithms Courses?

Guy Blelloch
Carnegie Mellon University

Why bother?

Parallelisms works even on modest machines

Hard to talk about scalability without parallelism

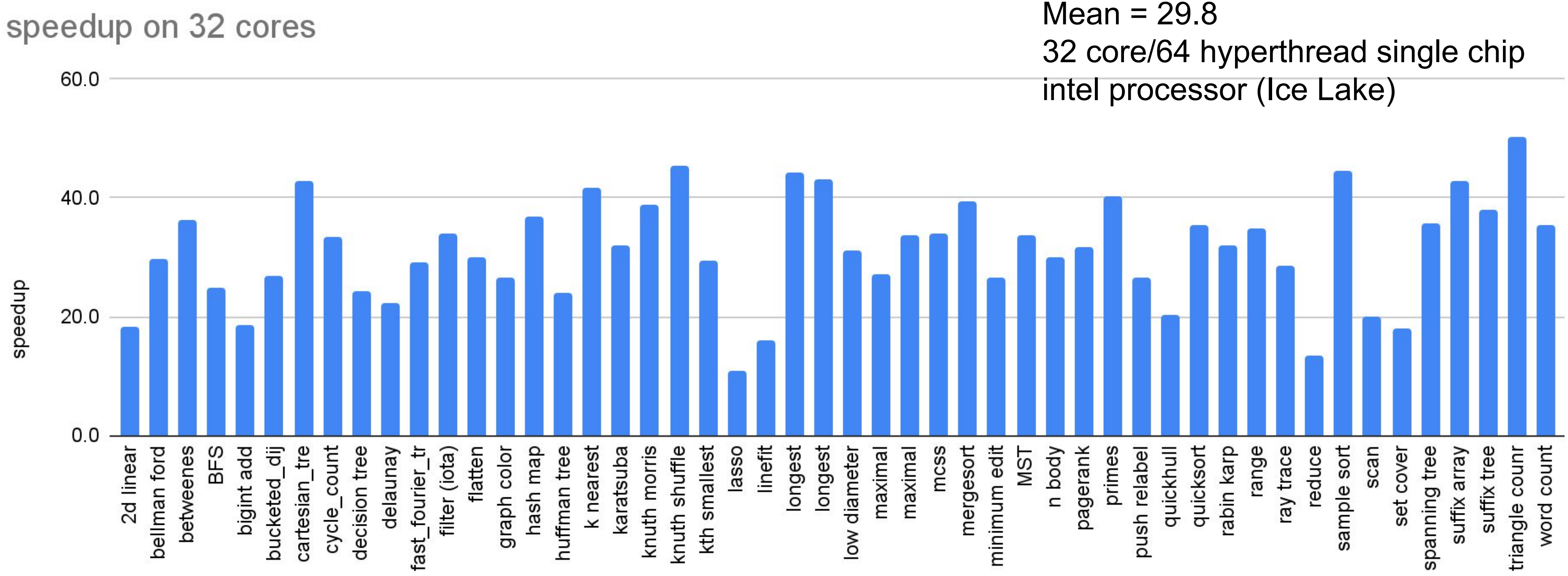
In many cases more natural and simple to think parallel

Selfish reason: To help Guy, Kunal, Yihan … get well educated graduate students

Lead to good efficiency?

Natural question: do ideas taught in corse help in practice

Mean = 29.8
32 core/64 hyperthread single chip intel processor (Ice Lake)



How hard is it?

We (at CMU) have been teaching parallelism in our intro (sophomore level) algorithms + data structure course (500 students/year) for 12 years.

- Not just a 2 lecture add on

Outline: Will give an overview of what we do and lessons learned.

Feedback: Would love to know any blocking factors for others, or what others are doing, and other ideas.

Philosophy

At the right level of abstraction there is no or little fundamental difference between parallel and sequential data structures and algorithms

- Need to recognize when parallelism is already present
- Need to understand how to make solutions more parallel
- Difference is only in costs and some added techniques

Emphasis on “**Parallel Thinking**”: i.e., ideas that are ”natural” and transcend the particular model.

Pros and Cons

Pros:

- Do not have to diverge much from standard algorithms course structure
- Emphasizes that parallelism is not esoteric
- Learn to think about parallelism abstractly and naturally

Cons:

- Requires revising an algorithms course throughout (significant work for faculty), although by how much is up to the instructor.

Course Outline

- Cost models and analysis
- Techniques: D&C, brute force, …
- Algorithms on sequences: merging, mergesort, scan, ..
- Randomization: quicksort, …
- BSTs and Priority Queues
- Graph algorithms (search, shortest paths, MSTs, …)
- Dynamic programming

Traditional at this level of detail

Parallel Model

(e1 || e2) : evaluate e1 and e2 in parallel, returning a pair when both are done

Sequence operations: e.g.

- Map : [f(x) : x in S]
- Tabulate : [f(i) : 0 <= i < n]
- Filter : [x in S | f(x)]

All calls to f are in parallel, returning a sequence

Avoid race conditions (schedule/interleaving) does not matter

Sidebar: easy to e.g. express parallelism in python

Even if implementation does not support it, e.g.,

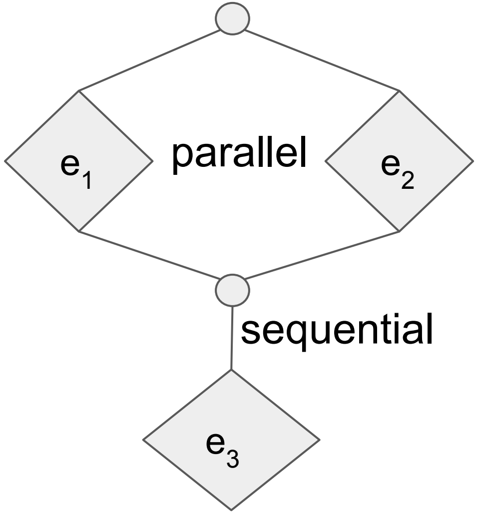
```
for i in range(n)        ⇨        [f(x) for x in S]
  r[i] = f(S[i])

r = 0
for x in S:              ⇨        reduce(add, S)
  r = r + x
```

Cost Model : Work and Span

Work: Total number of operations (like sequential time)
Span: Critical path (assuming unbounded number of processors)

	Sequential composition	Parallel composition
Work	add	add
Span	add	max



No PRAM

Cost model

Priorities:

- Most important is to reduce the work
- Then to reduce the span - ignore logarithmic factors.

Parallelism = Work/Span (roughly # of processors can make use of)

Some discussion of scheduling, leading to:

Time = Work/Processors + O(Span). for “greedy“ schedulers

Recurrences and Big-O

Same as sequential algorithms but for work and span instead of time.

E.g. for mergesort:

- $W(n) = 2W(n/2) + O(n) = O(n \log n)$ using sum over recursive calls
- $S(n) = S(n/2) + O(\log n) = O(\log^2 n)$ using max over recursive calls

Assuming for merge $W(n) = O(n)$ and $S(n) = O(\log n)$

Techniques

- Brute force
- Divide-and-conquer
- Contraction
- Graph search
- Dynamic programming

Techniques

- Brute force (note that naturally parallel)
- Divide-and-conquer (naturally parallel, and use more aggressively)
- Contraction (not usually covered in sequential algorithms)
- Graph search (BFS is parallel but not DFS)
- Dynamic programming (view more abstractly as a DAG)

Divide and Conquer

Same as with sequential, but:

- Recurrences for span
- More emphasis (use where iteration is used sequentially), e.g., reduce, parenthesis matching, MCSS
- Often need to strengthen induction

Divide and Conquer: reduce

Taking the sum with respect to a binary associative function f

```
reduce f identity S =
   Case splitMid(x) of
      Empty => return identity
    | Single(x) => return x
     | Pair(L, R) => return  f(reduce f identity L || reduce f identity R)
```

$W(n) = 2W(n/2) + W_f = O(n)$ assuming $W_f = S_f = O(1)$

$S(n) = S(n/2) + S_f = O(\log n)$

Would normally do iteratively in sequential setting.

Divide and Conquer: mergeSort

Taking the sum with respect to a binary associative function f

```
mergesort S =
   case splitMid(x) of
      Empty => return []
    | Single(x) => S
     | Pair(L, R) => return  merge(mergesort L || mergesort R)
```

$W(n) = 2W(n/2) + O(n) \quad = O(n \log n)$
$S(n) = S(n/2) + \log n = O(\log^2 n)$

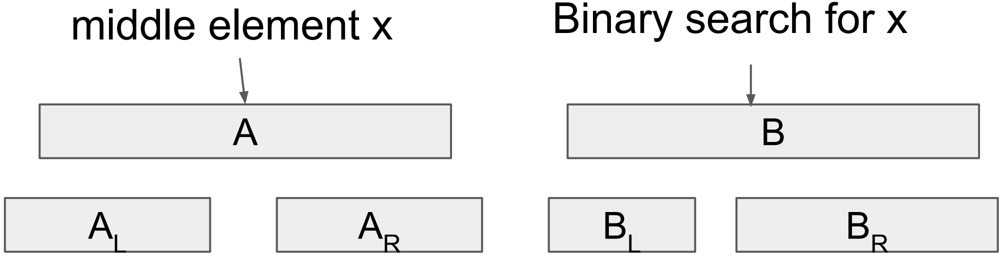
Assuming for merge: $W(n) = O(n)$, $S(n) = O(\log n)$

Divide and Conquer: mergeSort

Or:

Mergesort S = reduce(merge,[], [[x] : x in S])

Divide and Conquer: Merging



$$\text{merge}(A_L, B_L) \;||\; \text{merge}(A_R, B_R)$$

If use dual binary search:

$$W(n) = 2W(n/2) + O(\log n) \quad = O(n)$$
$$S(n) = S(n/2) + O(\log n) = O(\log^2 n)$$

Very different from sequential algorithm.
Can get $S(n) = O(\log n)$ with more advanced approach.

Divide and Conquer: Parenthesis Matching

Do these parentheses match: (()(()()(()))(()()))

Strengthen the induction to return number of unmatched parentheses on left and right:

```
parenMatch’ S=
   Case split_mid(x) of
      Empty => return (0,0)
    | Single(x) => if x = ‘(‘ then return (0,1) else return (1,0)
    | Pair(L,R) =>
         ((la,lb), (ra,rb)) =  (pareMatch’ L, parenMatch’ R)
         if (lb > ra) then return (la, lb-ra + rb)
         else return (la + ra - lb, rb)
```

```
parenMatch S =
  (a,b) = ParenMatch’ S
  return  (a = 0) and (b = 0)
```

Technique: Contraction

Basic structure:

- Reduce problem to one that is a constant factor smaller
- Recurse on the problem
- Use result to solve the problem

Some applications: Scan, kthSmallest, Graph Contraction

Not usually covered in intro algorithms/data structure course, but important for parallelism.

Contraction Example: Scan (prefix sums)

For binary associative operator f, return “running sum” of previous elements

e.g.:   Scan + 0 [2, 1, 3, 1, 2] -> [0, 2, 3, 6, 7], 9

Seems inherently sequential

Parallel version has many applications:
E.g.: filter, flatten, carry propagation, skyline, …

Scan: using contraction

```
scan(f, identity, S) =
  If |A| = 0 then return ([], identity)
  else if |A| = 1 then return ([identity], S[0])
  else
    pairSums = [ f(A[2i],A[2i+1]) : 0 <= i < |S|/2 ]     // pairwise sum the elements
    (r,s) = scan(f, identity, pairSums)                   // recurse
    return [if (i%2 =0) then r[2i] else r[2i] + a[i]]     // put together results
```

$W(n) = W(n/2) + O(n) = O(n)$

$S(n) = S(n/2) + O(1) = O(\log n)$

Assuming f is constant work:

Randomized algorithms

Analyze:

- expectation for work (e.g. $E[W_{quicksort}(n)] = O(n \log n)$)
- high probability bounds for span (e.g. $S[W_{quicksort}(n)] = O(\log^2 n)$ w.h.p.)

Problem is there is no equivalent of linearity of expectation for maximum.

Example of students taking an exam.

Randomized algorithms: kthSmallest

```
kthSmallest(A, k) =
    p = pick an element of A uniformly at random
    L = [x in A | x < p]
    R = [x in A | x > p]
    if (k < |L|) then return kthSmallest(L, k)
    else if (k < |A| - |R|) then return p
    else return kthSmallest(R, k - (|A| - |R|))
```

Show that number of recursive calls is O(log n) with high probability
Also work is O(n) in expectation.

Sidebar: Deteministic kthSmallest

$W(n) = W(7n/10) + W(n/5) + O(n) = O(n)$

$S(n) = S(7n/10) + S(n/5) + O(1) = O(n^{.82..})$

Parallelism = $O(n^{.18})$

Randomized algorithms: Quicksort

```
quicksort(A) =
    p = pick an element of A uniformly at random
    L = [x in A | x < p]
    R = [x in A | x > p]
    (L’, R’) = (quicksort(L) || quicksort(R))
    return L ++ [p] ++ R
```

Work = O(n log n) in expectation
Span = O($\log^2$ n) with high probability

Binary search trees:

Traditionally teach insertion, deletion, find on sets or dictionaries

None of these have parallelism.

Extend to support:

- Filter, reduce, multi-insert, …
- Union, intersection, ..

Allows for bulk operations on sets and dictionaries (Parallel Thinking)

Can all be built up from “join”, and are highly parallel

Join for treaps

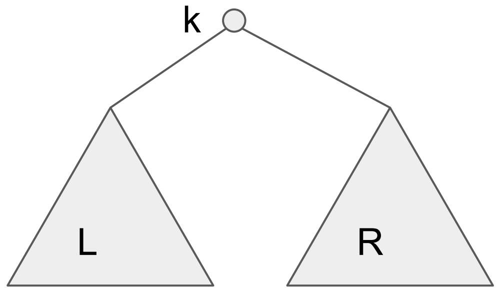
```
join(A, key, B) =
  if (p(k(A)) < p(key) and p(k(B)) < p(key)) then
    return Node(A ,key ,B)
  else if (p(k(A)) > p(k(B))) then
    return Node(L(A), key, join(R(A) ,k(A), B))
  else
    return Node(join(A, k(B), L(B)), key, R(B))
```

joinPair(A,B) and
split(A, k) => (AL, ?, AR)
can easily be built on this

W(n) = S(n) = O(log n) whp

Binary search trees: Filter example

```
filter f A =
   case A of
      Leaf => A
    | Node(L,k,R) =>
         (L', R') = (filter f L, filter f R)
         if (f(k)) return join(L',k,R')
         else return joinPair(L', R')
```



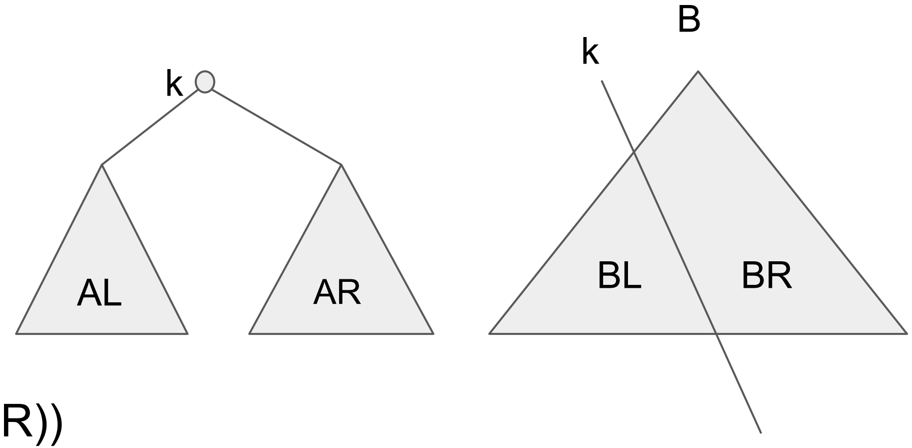
Work = $O(n)$
Span = $O(\log^2 n)$ whp

Join handles the rebalancing.
We use treaps for balancing, but could use AVL trees or Red Black trees.

Binary search trees: Union

```
union A B  =
   case (A,B) of
      (Leaf,_) => B
     | (_, Leaf) => A
     | (Node(AL,k,AR),_) =>
        (BL,_,BR) = split(B, k)
        (L', R') = (union(AL,BL) || union(AR,BR))
        return join(L',k,R')
```

Work = $O(m \log(n/m + 1))$
Span = $O(\log^2 n)$



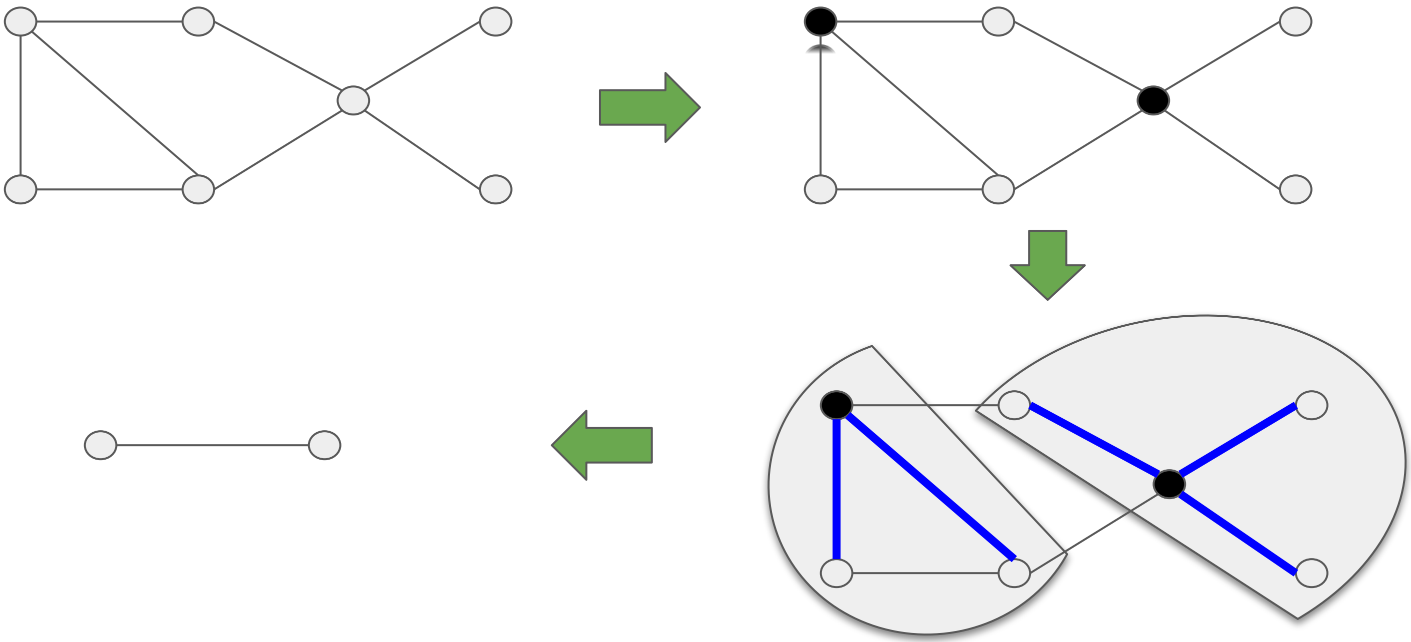
Graphs:

30% of course is on graphs

Some key points:

- BFS is parallel, span is O(d log n), where d is the diameter of the graph
- DFS and Dijkstra are sequential
- Bellman Ford is parallel: O(mn) work and O(n log n) span.
- Johnson is parallel
- Use graph contraction for connectivity and Min Spanning Tree MST)
  MST uses Boruvka's algorithm, but we also mention Kruskal and Prim

Graphs: Example of star contraction



Graphs: Star contraction

In one round remove at least ¼ of the vertices in expectation.

Therefore finishes in O(log n) rounds with high probability.

Total work is O(m log n)

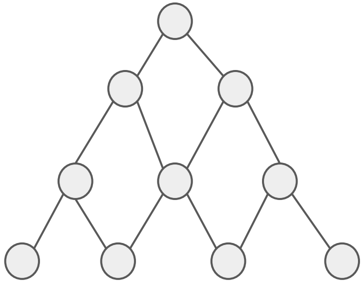
Used for both Connectivity and Boruvkas MST algorithm

Dynamic Programming:

To expose parallelism, we raise level of abstraction.

DP is recursion with sharing. Sharing can be viewed as a DAG.

Recursion Tree -> Dag



Calculate total work across all recursive calls.

Calculate longest path in DAG (same as span for non-shared solution)

Dynamic Programming:

Some examples we cover:

- Subset sum: $W(n,k) = O(nk)$, $S(n, k) = O(n)$
- Minimum edit distance: $W(n,m) = O(nm)$, $S(n,m) = O(n + m)$
- Optimal BST: $W(n) = O(n^3)$, $S(n) = O(n \log n)$
- Matrix chain Product: $W(n) = O(n^3)$, $S(n) = O(n \log n)$

Discuss bottom up and top down approaches.

Lessons Learned

- **Deterministic** parallelism not difficult for the students to understand.
  - They get the hang quickly.
- List of “new” ideas that need to be covered, and which are easier/harder
- resistance by faculty to changing existing courses
  - faculty often not comfortable with topic
- best to do along with considering whole curriculum

Have not yet done a careful study

Lessons Learned: effect on whole curriculum

- ideally requires changes in prerequisites
  - parallel thinking earlier
  - programming with comprehensions and functional style earlier
  - tail bounds in probability
- changes to follow up courses
  - how to cover parallelism in Senior level algorithms class?
  - where to cover concurrent and distributed algorithms?
- important to have complementary course on systems issues
  - gpus, distributed computing, locks, …

Why are sequential data structures and algorithms so successful?

- abstract away from detail
- broadly applicable
- general techniques
- simple cost model
- easy to program
- elegant and often simple
- help understand scaling, i.e., big-O
- lead to good efficiency in practice
- interesting theory

Conclusions

Parallel and sequential algorithms can be integrated such that students

- consider parallelism throughout
- learn “ standard” algorithmic topics, e.g. D&C, DP, mergesort, quicksort, BFS, DFS, …
- learn to think parallel, e.g., ask if an algorithm has parallelism, or if and how it can be added

Including in an existing UG curriculum has challenges