

Are Parallel Algorithms Ready for Prime Time

Guy Blelloch
Carnegie Mellon University

Are (sequential) Algorithms “Prime Time”?

First hit on google search “core computer science concepts”:



40 Key Computer Science Concepts Explained in Layman’s Terms

Core Concept #1 - Algorithms and Data Structures

1.1 Big O Notation

1.2 Sorting Algorithms

1.3 Recursion ...

1.4 Big Data

1.5 Data Structures

Core Concept #2 - Artificial Intelligence

2.1 Greedy Algorithms

Are (sequential) Algorithms “Prime Time”?

Most first year undergraduate CS curriculums cover:

1. Sorting (insertion, quick, merge, heap?)
2. Big-O, some introduction to recurrences
3. Binary trees, probably BFS
4. ...

Most programming languages have widely used libraries of algorithms:

1. STL in C++
2. Java collections
3. Python, the kitchen sink
4. ...

Used by just about all programmers

Why are algorithms so successful?

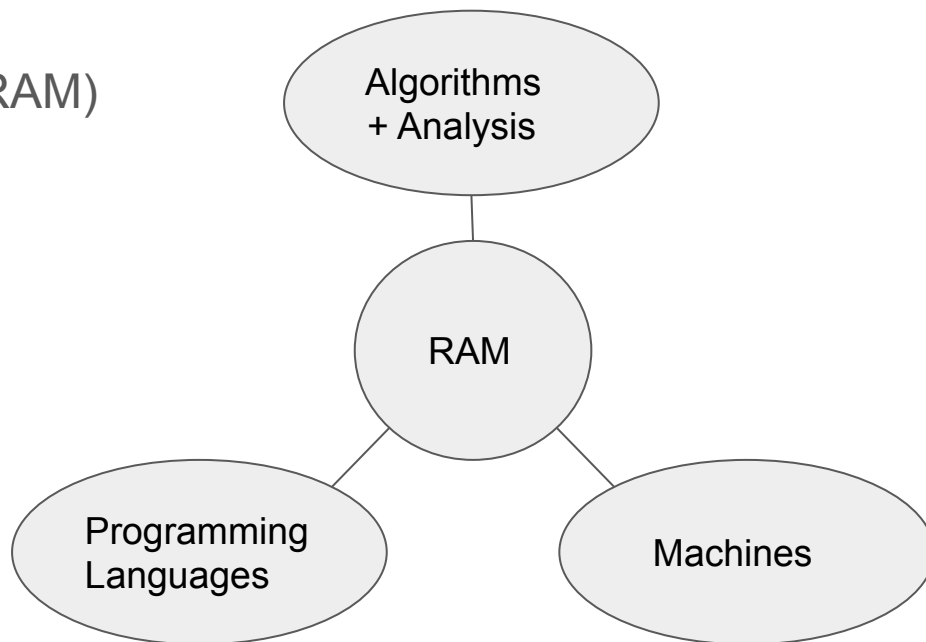
- **Broadly** applicable
- Support **general** techniques/abstractions
- Simple and **elegant**
- Are **easy** to program
- Rely on a **simple** cost model
- Lead to **interesting** theoretical questions
- Lead to good **efficiency** in practice

Why are algorithms so successful?

The **random access machine** (RAM)

A great bridging model

```
void DFS(int v, graph G, bool* visited) {  
    visited[v] = true;  
    visit(v);  
    for (int u : G.adj[v])  
        if (!visited[u]) DFS(u, G, visited);  
}
```



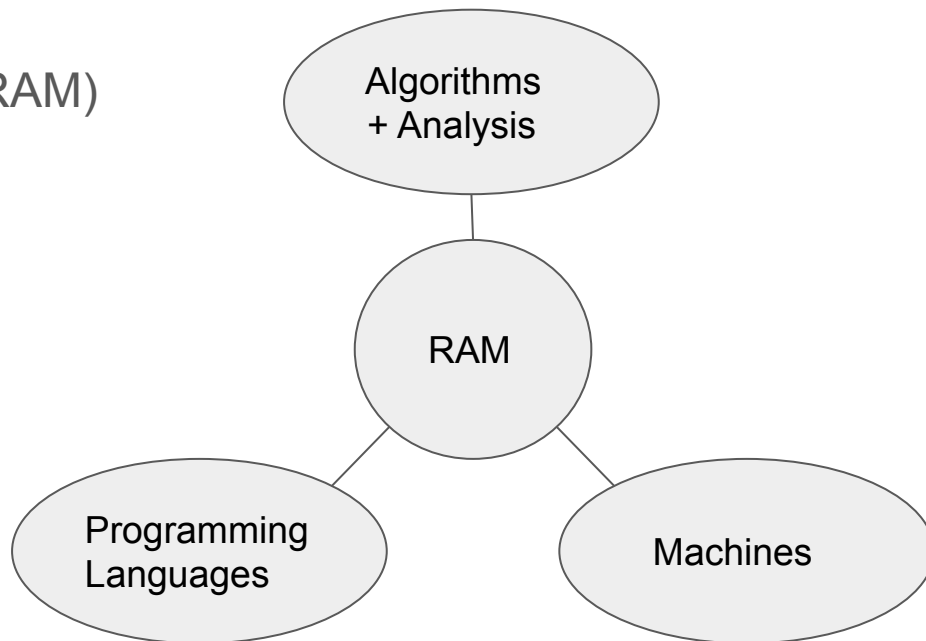
Why are algorithms so successful?

The **random access machine** (RAM)

A great bridging model

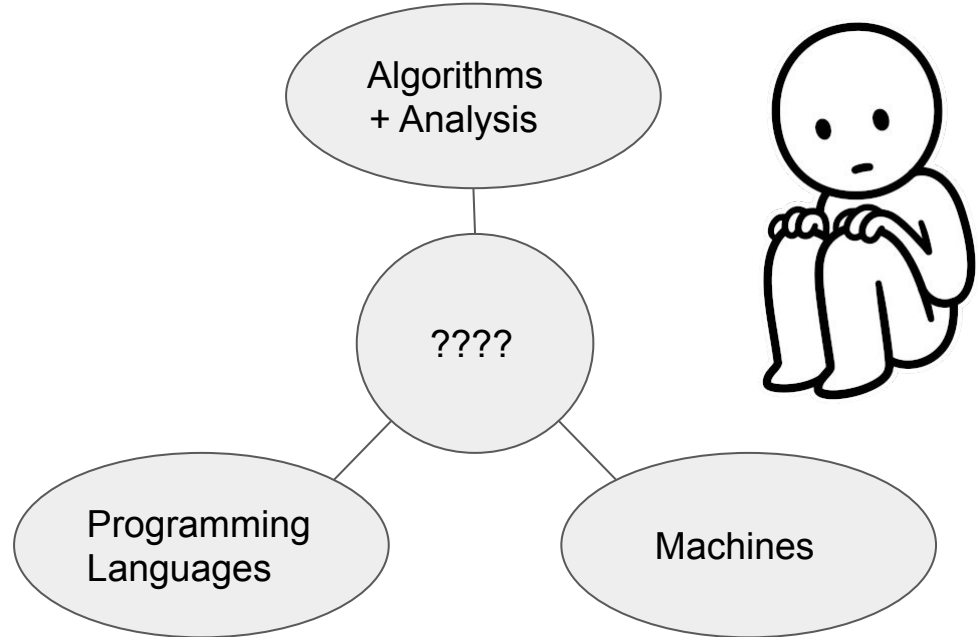
Not perfect:

- E.g. caches
- But there are natural extensions to handle that when needed



Can we get the same “ecosystem” for parallelism?

- **Broadly** applicable
- **general** techniques
- Simple and **elegant**
- **Easy** to program
- **Simple** cost model
- **Interesting** theory
- Good **efficiency** in practice



E.g. well suited to teach in first year UG classes instead of sequential algorithms

A brief history of parallel algorithms

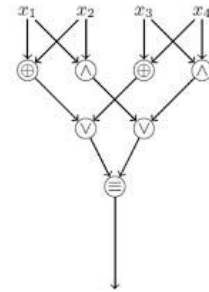
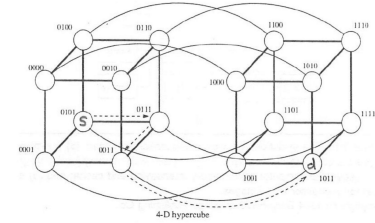
The 60s and 70s: early exploration

Network models (hypercube, butterfly, meshes, etc.), e.g. Batcher sort

- Too low level, not portable

Circuit models (Nick's class, the NC-hierarchy, P-complete problems)

- Not programmable
- NC ignores polynomial factors in work
- Some good parallel algorithms have polynomial dept



A brief history of parallel algorithms

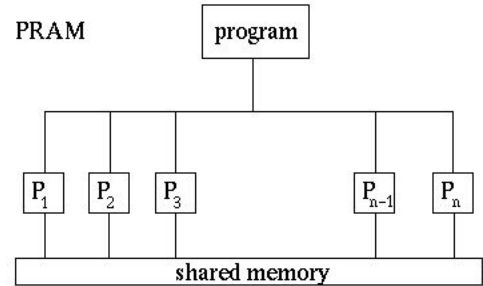
The 80s: The decade of the PRAM

100s of papers on the topic

Many cool ideas: Pointer jumping, random mate, random sampling, euler tour trees, scan, cascading, contraction

Very little code:

- Overly synchronous
- Not well suited for nested parallelism (e.g. parallel D&C)
- Ignores communication cost



A brief history of parallel algorithms

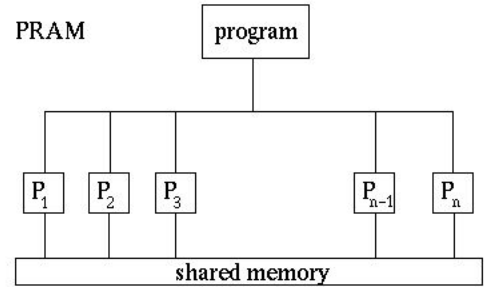
The 80s: The decade of the PRAM

100s of papers on the topic

Many cool ideas: Pointer jumping, random mate, random sampling, euler tour trees, scan, cascading, contraction

Real problem is assuming a **fixed number** of **synchronous** processors

- User needs to write their own “scheduler”



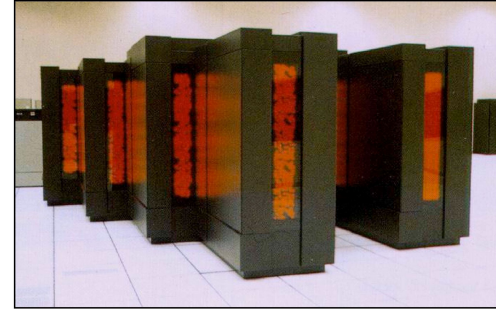
And parallel hardware



Cray
XMP
(1982)



Thinking
Machines
CM-2
(1987)



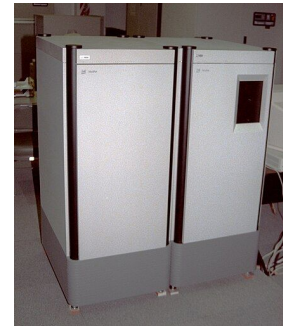
Thinking
Machines
CM-5
(1991)



Catech/Intel
Cosmic cube
(1984)



Intel
Paragon
(1993)



Maspar
MP-2
(1992)

First Parallel Desktop : The SugarCube (1987)



Intel SugarCube™ systems consists of the Cube Manager (bottom left), user terminal, and Cube (top left), which provides the computing power of 4 or 8 processing nodes.

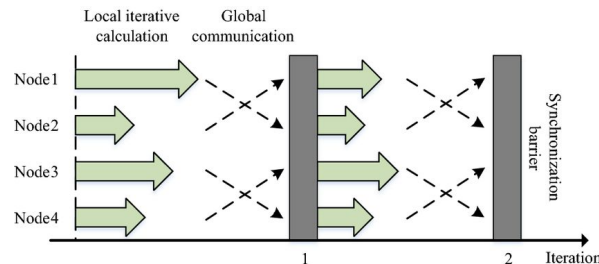
Prime time?

Although many excellent ideas:

- Million of \$s each (mostly government labs)
- Programming was very difficult
- SIMD/vector and/or distributed memory made it hard
- Communication was inadequate
- Theory (PRAM) did not match practice

A brief history of parallel algorithms

The 90s: The PRAM “gone afoul”



The \log^* failure: focus on non-robust details of the PRAM

Various more “realistic” models:

- BSP, LogP : account for communication, but too synchronous, and pain to design algorithms for
gap : number of instructions per word transferred
- asynchronous PRAM : suffer some of the same problems as PRAM

Nested parallel (fork/join, work-depth model) : will come back to

The High Cost of Low Communication (1995)

The cost of communication

Why does bandwidth effect the cost of software development?

Particularly for applications on irregular data

1. Locality complicates algorithms, languages and compilers.
2. Different algorithms for different data.
3. Complicates abstraction boundaries.

Claim: Harder to deal with locality than parallelism. (In a general way)

Can we build “scalable” machines with high bandwidth and low overhead communication?

(e.g. < 5 instructions per 1 word message)

gap

Conclusions

Non enough attention has been paid to the high software-development cost for *communication-impaired* machines.

Suggestion: The government should coordinate an effort to:

1. Standardized components/interfaces for low overhead, high throughput networks (cost sharing)
2. Develop hooks into standard microprocessors (coordinate with industry).

A brief history of parallel algorithms

The 00s+: special purpose

GPU Models:

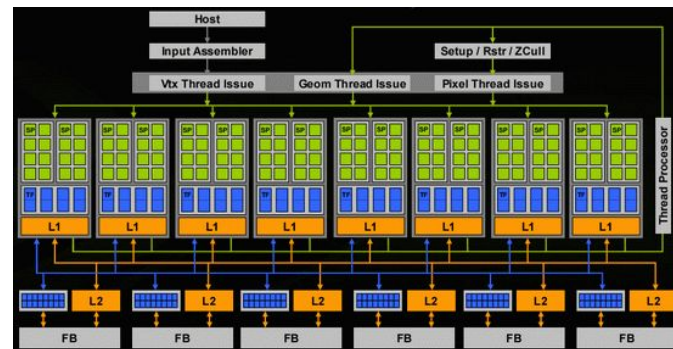
- To many details for a general model

Map-reduce models, e.g. MPC:

- Bulk synchronous is limiting
- Not shown to lead to efficient algorithms in practice

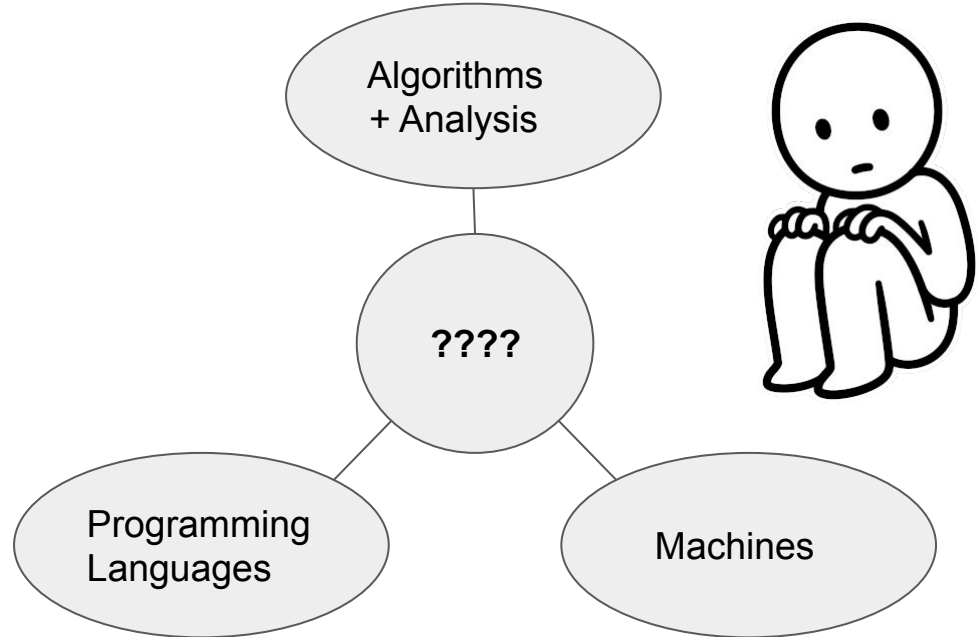
Domain specific models

- Not general



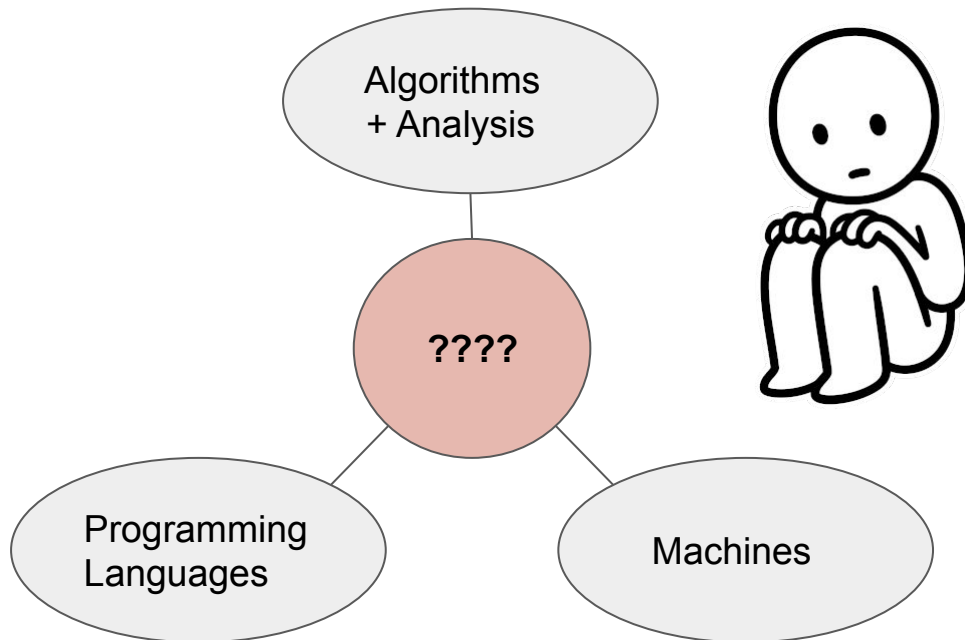
Can we get the same “ecosystem” for parallelism

- **Broadly** applicable
- **general** techniques
- Simple and **elegant**
- **Easy** to program
- **Simple** cost model
- **Interesting** theory
- Good **efficiency** in practice



Can we get the same “ecosystem” for parallelism

- **Broadly** applicable
- **general** techniques
- Simple and **elegant**
- **Easy** to program
- **Simple** cost model
- **Interesting** theory
- Good **efficiency** in practice



Nested Parallel Model

Nested fork-join parallelism

- Binary or multiway forking (only affects span)
- Costs in terms of Work and Span (Depth)
- Compose work with sum, Span with max

quicksort(A) =

 If ($|A| < 2$) return A

p = pick random pivot from A

$L = [x \text{ in } A \mid x < p]$

$R = [x \text{ in } A \mid x > p]$

$(L', R') = (\text{quicksort}(L) \parallel \text{quicksort}(R))$

 return $L \mathbin{++} [p] \mathbin{++} R$

$W(n) = O(n \log n)$ whp

$S(n) = O(\log^2 n)$ whp

Desiderata

Broadly Applicable

- Hardly know a clean parallel algorithm that does not fit the model

General Techniques

- Existing: Divide-and-conquer, dynamic programming, balanced trees, ..
- Newish: Contraction, random-mate, pointer doubling, scan (prefix sums)

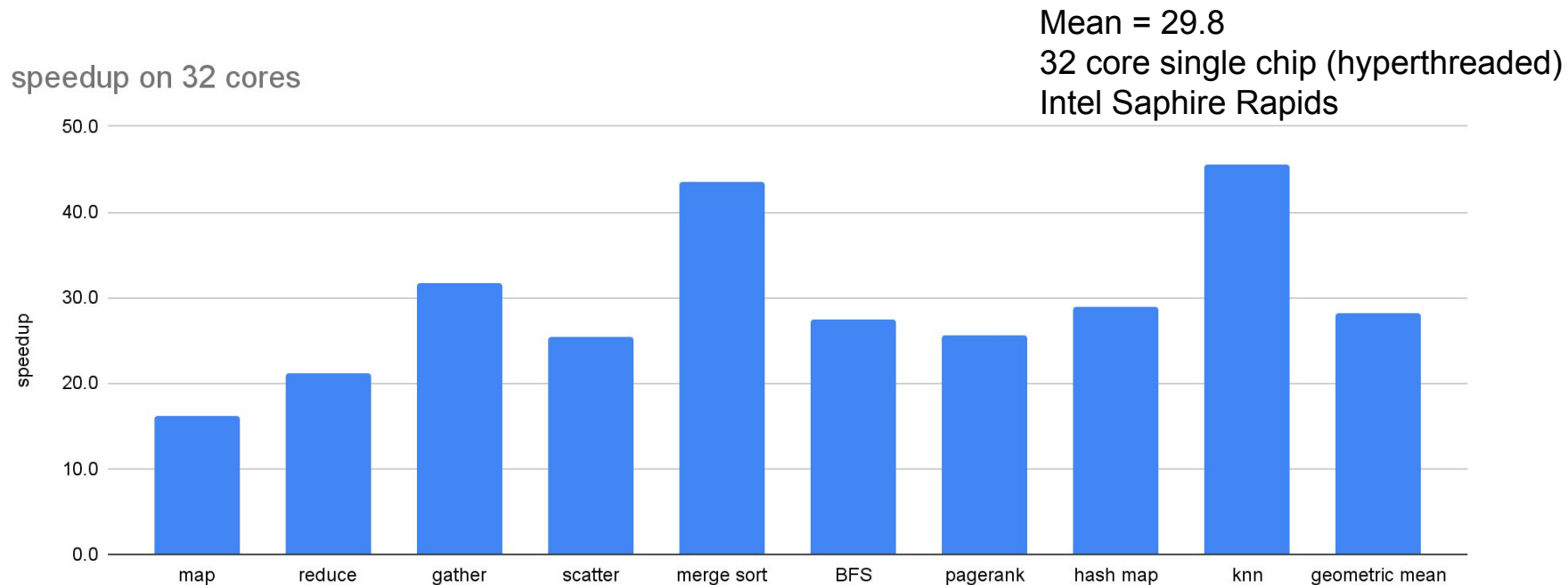
Interesting Theory

- Of course

Desiderada

Lead to good efficiency in practice?

Do multicore's work for parallelism?

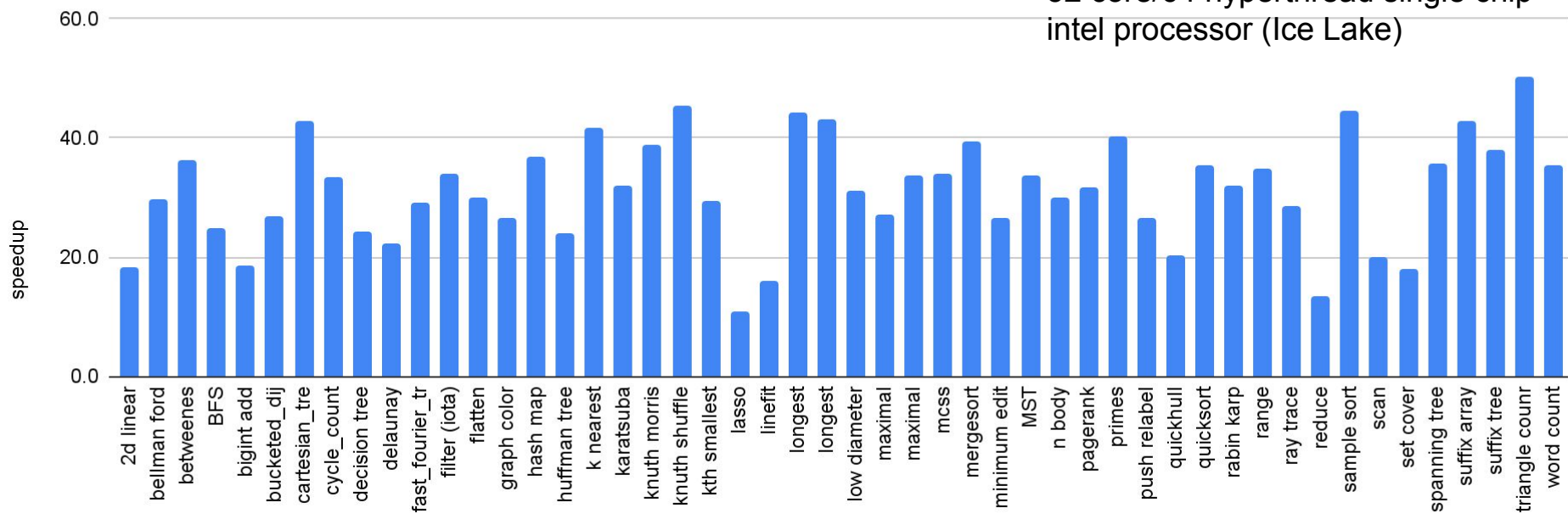


Parlaylib: Examples

speedup on 32 cores

Mean = 29.8

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



Graph algorithms results (SPAA18)

Problem	(1)	(72h)	(SU)	Alg.	Model	Work	Depth
Breadth-First Search (BFS)	576	8.44	68	–	TS	$O(m)$	$O(\text{diam}(G) \log n)$
Integral-Weight SSSP (weighted BFS)	3770	58.1	64	[42]	PW	$O(m)$ expected	$O(\text{diam}(G) \log n)$ w.h.p. [†]
General-Weight SSSP (Bellman-Ford)	4010	59.4	67	[38]	PW	$O(\text{diam}(G)m)$	$O(\text{diam}(G) \log n)$
Single-Source Widest Path (Bellman-Ford)	3210	48.4	66	[38]	PW	$O(\text{diam}(G)m)$	$O(\text{diam}(G) \log n)$
Single-Source Betweenness Centrality (BC)	2260	37.1	60	[30]	FA	$O(m)$	$O(\text{diam}(G) \log n)$
$O(k)$ -Spanner	2390	36.5	65	[89]	TS	$O(m)$	$O(k \log n)$ w.h.p.
Low-Diameter Decomposition (LDD)	980	16.6	59	[90]	TS	$O(m)$	$O(\log^2 n)$ w.h.p.
Connectivity	1640	25.0	65	[117]	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Spanning Forest	2420	35.8	67	[117]	TS	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
Biconnectivity	9860	165	59	[125]	FA	$O(m)$ expected	$O(\max(\text{diam}(G) \log n, \log^3 n))$ w.h.p.
Strongly Connected Components (SCC)*	8130	185	43	[23]	PW	$O(m \log n)$ expected	$O(\text{diam}(G) \log n)$ w.h.p.
Minimum Spanning Forest (MSF)	9520	187	50	[130]	PW	$O(m \log n)$	$O(\log^2 n)$
Maximal Independent Set (MIS)	2190	32.2	68	[22]	FA	$O(m)$ expected	$O(\log^2 n)$ w.h.p.
Maximal Matching (MM)	7150	108	66	[22]	PW	$O(m)$ expected	$O(\log^3 m / \log \log m)$ w.h.p.
Graph Coloring	8920	158	56	[59]	FA	$O(m)$	$O(\log n + L \log \Delta)$
Approximate Set Cover	5320	90.4	58	[25]	PW	$O(m)$ expected	$O(\log^3 n)$ w.h.p.
k -core	8515	184	46	[42]	FA	$O(m)$ expected	$O(\rho \log n)$ w.h.p.
Approximate Densest Subgraph	3780	51.4	73	[13]	FA	$O(m)$	$O(\log^2 n)$
Triangle Counting (TC)	–	1168	–	[119]	–	$O(m^{3/2})$	$O(\log n)$
PageRank Iteration	973	13.1	74	[31]	FA	$O(n + m)$	$O(\log n)$

Why is the performance good:

Memory Bandwidth (all machines with 2.5MHz processors)

Recall: **gap** = number of 8-byte words loaded/processor/cycle

Intel Haswell (2016), 18 core (1 chip), 80GB/sec: **gap** = 4.5

Cascade Lake (2018), 28 core (1 chip), 140GB/sec: **gap** = 4

Intel Ice Lake (2021), 32 core (1 chip), 200GB/sec: **gap** = 3.2

Intel Sapphire Rapids (2023), 32 core (1 chip), 300GB/sec: **gap** = 2.2

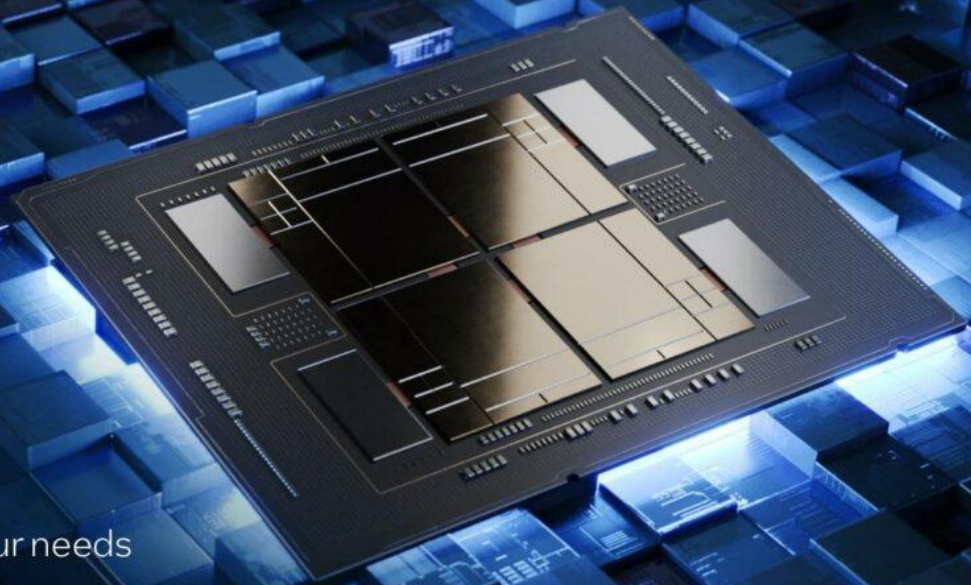
Speedup of Reduce from: 9x -> 12x -> 15x -> 22x



First & only x86 CPU with HBM



Choose the right memory configuration for your needs



Memory Modes

64GB
HBM2e
4 stacks of 16GB

~1TB/s
Memory Bandwidth

>1GB
HBM per Core

HBM Only
Bootable from HBM
No code change




HBM Flat
2 Memory Regions
SW Optimization Needed




HBM Caching
HBM as cache for DDR
No code change





Under Embargo until November 9th, 2022, 6am PT

Desiderata

Simple and **elegant and Easy** to program?

- BFS
- Merging
- Union of two balanced BSTs

Breadth First Search

```
void BFS(int start, sequence<sequence<int>& G) {
    auto visited = tabulate<std::atomic<bool>>(G.size(), [&] (long i) {
        return (i==start) ? true : false; });

    sequence<int> frontier(1, start);
    while (frontier.size() > 0) {
        auto ngh = flatten(map(frontier, [&] (vertex u) {
            return G[u];}));

        frontier = filter(ngh, [&] (int v) {
            return (!visited[v]) && visited[v].CAS(false, true);});
    } }
```

$$W(n,m) = O(m)$$

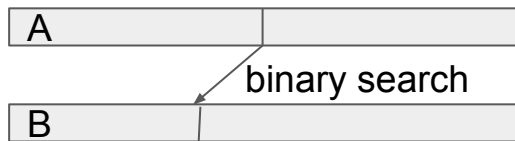
$$S(n,m) = O(d \log n)$$

n vertices, m edges, diameter d

Merging (Divide and Conquer)

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

```
void merge(Slice A, Slice B, Slice R, F f) {
    long nA = A.size(); long nB = B.size(); long nR = nA + nB;
    if (nR < Threshold) std::merge(A.begin(), A.end(), B.begin(), B.end(), R.begin(), f);
    else if (nA == 0) copy(B, R);
    else if (nB == 0) copy(A, R);
    else {
        long mA = nA / 2;
        long mB = std::lower_bound(B.begin(), B.end(), A[mA], f);
        long mR = mA + mB;
        par_do([&]() { merge(A.cut(0, mA), B.cut(0, mB), R.cut(0, mR), f);},
            [&]() { merge(A.cut(mA, nA), B.cut(mB, nB), R.cut(mR, nR), f);});
    }
}
```



Union of two balanced BSTs (e.g. Red-Black)

```
node* union(node* A, node* B) {  
    if (!A) return B;  
    if (!B) return A;  
    auto [leftA, rightA] = split(A, B->key);  
    auto [left, right] = par_do_pair(  
        [&] {return union(leftA, B->lc);}  
        [&] {return union(rightA, B->rc);});  
    return join(left, B->key, right);  
}
```

Many other examples:

<https://github.com/cmuparlay/parlaylib>, in examples directory (search **parlaylib**)

2d_linear_program.h	find_if.h	longest_common_prefix.h	radix_tree.h
3d_range.h	flatten.h	longest_repeated_substring.h	range_min.h
bellman_ford.h	graph_color.h	low_diameter_decomposition.h	ray_trace.h
betweenness Centrality.h	hash_map.h	maximal_independent_set.h	rectangle_intersection
BFS.h	huffman_tree.h	maximal_matching.h	reduce.h
BFS_ligra.h	karatsuba.h	mcSS.h	samplesort.h
bigint_add.h	kcore.h	mergesort.h	scan.h
box_kdtree.h	kmeans_pp.h	minimum_edit_distance.h	set_cover.h
bucketed_dijkstra.h	knn.h	nbody_fmm.h	spanning_tree.h
cartesian_tree.h	knuth_morris_pratt.h	oct_tree.h	spectral_separator.h
cycle_count.h	knuth_shuffle.h	pagerank.h	suffix_array.h
decision_tree_c45.h	kruskal.h	primes.h	suffix_tree.h
delaunay.h	kth_smallest.h	push_relabel_max_flow.h	tokens.h
fast_fourier_transform.h	lasso_regression.h	quickhull.h	triangle_count.h
filter.h	le_list.h	quicksort.h	word_counts.h
filter_kruskal.h	linefit.h	rabin_karp.h	

Conclusion:

Are Parallel Algorithms ready for prime time?

- **Broadly** applicable
- **General** techniques
- Simple and **elegant**
- **Easy** to program
- **Simple** cost model
- **Interesting** theory
- Good **efficiency** in practice
- Easy to debug

Yes for multicores.

**Not yet for other models of parallelism
(GPUs, MPC, PIM, Vector Units, ...)**

**This is OK. Good to separate the part
we can move to e.g. an UG curriculum.**

Purpose of asymptotic (big-O) analysis:

+ **yes**

1. **Abstraction** : avoid details
2. **Guidance** : towards a good algorithm
3. **Scalability** : how will cost grow with size
4. **Justify** : algorithms, data structures and techniques

- **no**

1. **Runtime** : how fast will it run on my x247mpq-7rl-v3
2. **Fine Tuning** : lets get the last 10%
3. **Fine Details** : lets strip a $\log^* n$ off of an n^2 bound (my opinion)

What about locality

Achievable:

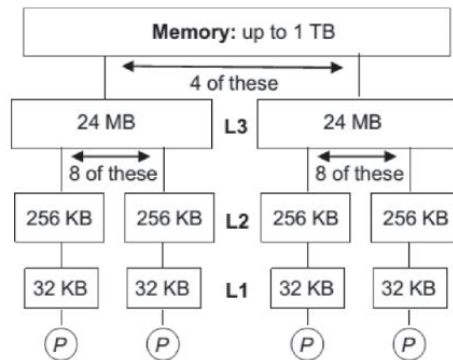
There is an inherent “sequential” left-to-right order.

Analyze cache cost in this order.

E.g. block recursive matrix multiply has same cost as sequentially

This leads to (provably) good behavior when simulated on various parallel cache hierarchies with shared and distributed caches, e.g.:

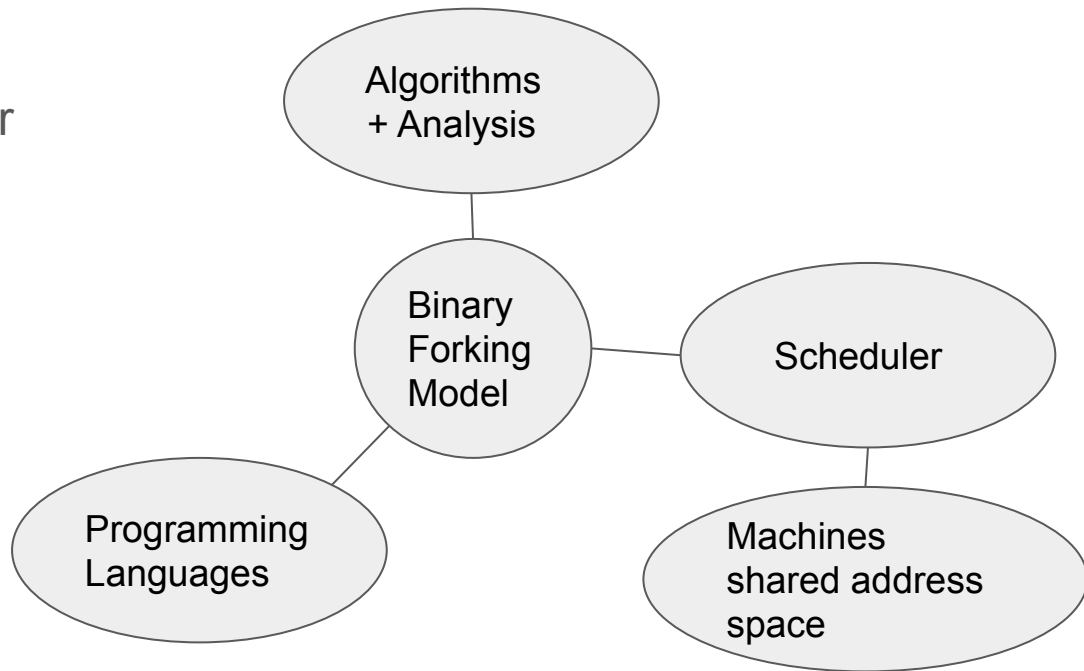
- Shared caches : use priority first scheduling
- Distributed Caches : use work-stealing
- Hierarchical caches : use space-bounded schedulers



Can we get the same “ecosystem” for parallelism

The binary-forking model

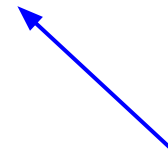
A great bridging model, at least for some class of machines.



Quicksort

```
void quicksort(slice In, slice Out, Comp f, bool inplace) {  
    long n = In.size();  
    if (n < Threshold) {  
        std::sort(In.begin(), In.end(), f);  
        if (!inplace) copy(In, Out);  
    } else {  
        double p = In[n/2];  
        auto sizes = bucket_by(In, Out, [] (auto k) {return f(k,p) ? 0 : f(p,k) ? 2 : 1;}, 3);  
        long l = sizes[0]; long h = sizes[0] + sizes[1];  
        par_do([&]() {quicksort(Out.cut(0, l), In.cut(0, l), f, !inplace);},  
            [&]() {quicksort(Out.cut(h, n), In.cut(h, n), f, !inplace);});  
        if (inplace) copy(Out.cut(l,h), In.cut(l,h));  
    }  
}
```

$W(n) = O(n \log n)$ w.h.p
 $S(n) = O(\log^2 n)$ w.h.p.



Partition into <, =, >

Education

We have been teaching this at CMU for almost 10 years now (started in 2012).

All our sophomores take a course “parallel and sequential data structures and algorithms” that teaches in this style.

Teach all the standard ideas + parallelism: D&C, DP, big-O, recurrences, DFS, BFS, Dijkstra's, ...

Parallelism is not hard for them.

What about other types of machines?

GPUs : becoming more like CPUs (perhaps they will become the same)

Distributed memory: seems hard to get General purpose clean model, but having a shared address space should be fine. Race free programs do not need cache coherence (flush when needed).

Processing in memory: some recent work

Conclusions

Question: can parallel algorithms/analysis replace sequential algorithms/analysis, or ideally be part of the same “ecosystem”?

Binary forking model is a step towards the goal:

- Integrates well with sequential algorithms
- Can incorporate locality
- Simple code, and fast implementations

But some caveats

- Does not cover all machines
- Getting community buy in to parallelism is hard

Can we get the same “ecosystem” for parallelism

Measures of success:

- Every undergraduate data structures and algorithms course covers parallel algorithms throughout.
- All CS professionals know a collection of parallel techniques and algorithms
- All mainstream languages properly support parallelism
- Most library implementations are parallel
- Algorithms remain simple
- **Parallel machine architecture helps simplify algorithm design**

Word Counts

```
auto wordCounts(charseq const &s) {  
    auto str = parlay::map(s, [] (char c) {return std::isalpha(c) ? c : 0;})  
    auto words = parlay::tokens(str, [] (char c) {return c == 0;});  
    return parlay::count_by_key(words);  
}
```

$$W(n) = O(|s|)$$
$$S(n) = O(|s|^{1/2})$$

Declares whitespace




Breadth First Search

```
vSequence BFS(vertex start, const Graph &G) {  
    size_t n = G.numVertices();  
    vSequence parent(n, -1);  
    parent[start] = start;  
    auto frontier = lgra::vertex_subset(start);  
  
    while (frontier.size() > 0)  
        frontier = lgra::edge_map(frontier,  
            [&] (vID v) { return parent[v] == -1;},  
            [&] (vID u, vID v) { return CAS(parent[v], -1, u);});  
    return parent;  
}
```

$W(n,m) = O(m)$
 $S(n,m) = O(d \log n)$
n vertices, m edges, diameter d

Maps over out-edges of each
vertex in the frontier

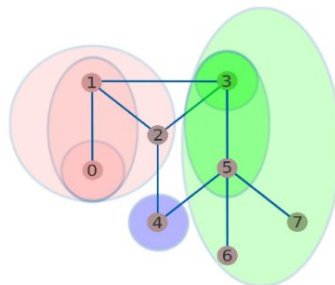


Graph Connectivity

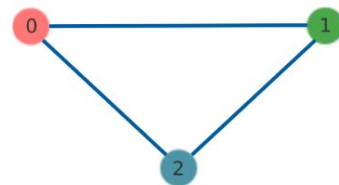
```
vSequence Connectivity(Graph& G) {  
    size_t n = G.n;  
    vSequence clusters = LDD(G);  
    long num_clusters = RelabelIds(clusters);  
    auto [G_clusters, flags, mapping] = Contract(G, clusters, num_clusters);  
    if (G_clusters.m == 0) return clusters;  
    auto new_labels = Connectivity(G_clusters, beta, level + 1);
```

```
    parallel_for(0, n, [&] (size_t i) {  
        vtxid cluster = clusters[i];  
        vtxid gc_cluster = flags[cluster];  
        if (gc_cluster != flags[cluster + 1])  
            clusters[i] = mapping[new_labels[gc_cluster]];  
    });  
    return clusters;  
}
```

$W(n,m) = O(m)$ whp
 $S(n,m) = O(\log^2 n)$ whp
n vertices, m edges, diameter d



(a) graph decomposition



(b) contracted graph