Is Asymptotic Cost Analysis Useful for Developing Practical Parallel Algorithms?

Guy Blelloch
Carnegie Mellon University

Contributions from many others including: Daniel Anderson, Laxman Dhulipala, Yan Gu, Julian Shun, Yihan Sun, ++

A quick google search on “core computer science concepts”:

CARLCHEO.com
Tech. Software. Computer Tips.

40 Key Computer Science Concepts Explained In Layman’s Terms

Carl Cheo Follow me on Twitter here.

Core Concept #1 - Algorithms and Data Structures

- 1.1 Big O Notation
- 1.2 Sorting Algorithms
- 1.3 Recursion …

A defining aspect of our field

Purpose of asymptotic (big-O) analysis:

+ yes

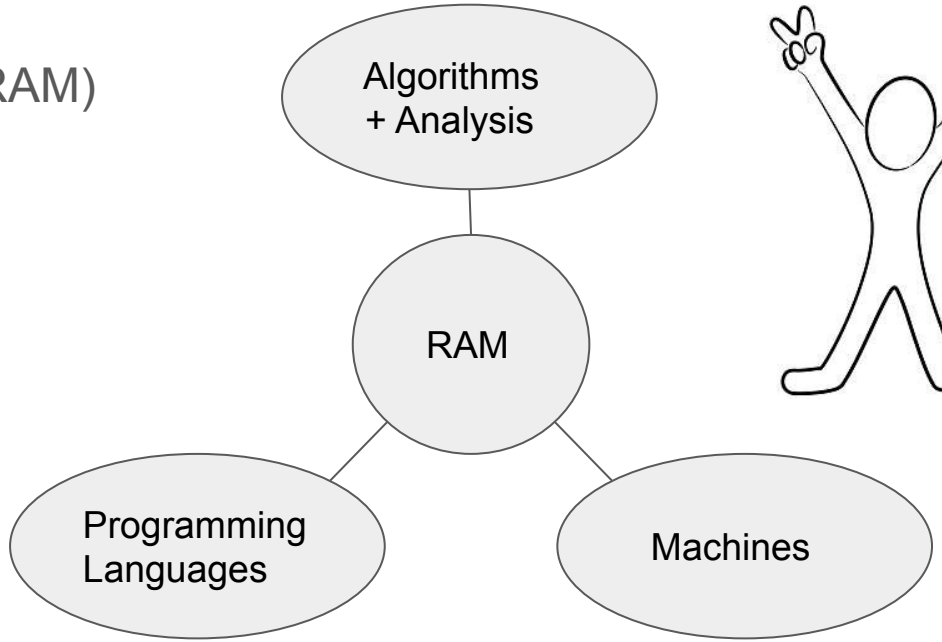
1. **Abstraction** : avoid details
2. **Guidance** : towards a good algorithm
3. **Scalability** : how will cost grow with size
4. **Justify** : algorithms, data structures and techniques

- 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)

Why is algorithm analysis so “successful”?

The **random access machine** (RAM)
A great bridging model,
i.e., a great abstraction.



The good, the bad and the ugly

Good : sorting, BFS, DFS, balanced trees, Dijkstra's, DP, hash tables, Delaunay triangulation, edit distance, ...

Bad: Matrix multiplication

Ugly: SAT solving

```
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);
}
```

- Simple analysis
- Simple code
- Runs fast

Side bar: Adding locality (and other features)

The IO-model and cache-oblivious algorithms
No change to code, just analysis

- Block size B, Cache size M
- Matrix multiply triply nested loops: $O(\frac{n^3}{B})$
- Matrix multiply block recursive: $O(\frac{n^3}{B\sqrt{M}})$

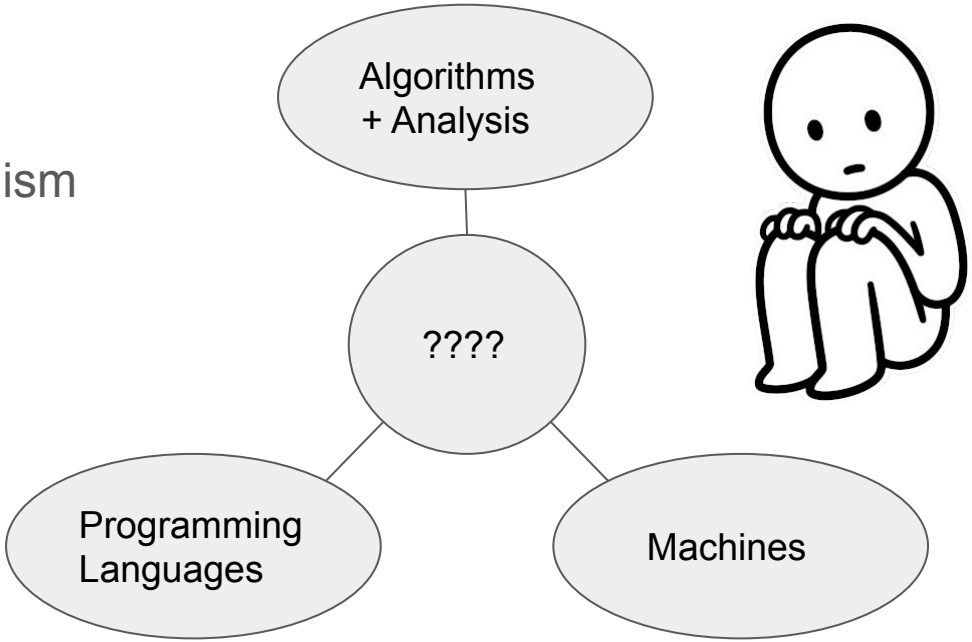
Many algorithms have been analyzed in the model, and leads to very cache friendly algorithms.

Can add read-write asymmetry, and other factors

Can we get the same “ecosystem” for parallelism

The ???
A great bridging model,
i.e., a great abstraction for parallelism

or give up?

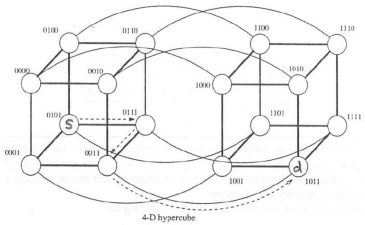


Can we get the same “ecosystem” for parallelism

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**

A brief history of parallel models



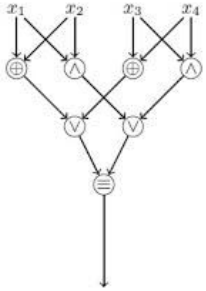
The 60s and 70s: early exploration

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

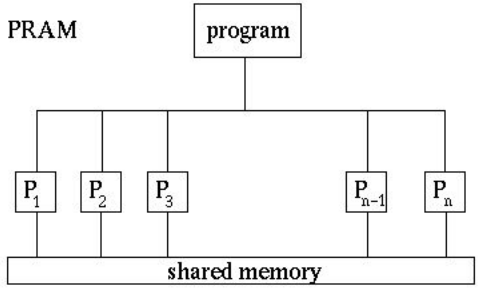
- Too low level, not portable

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

- Not programmable
- NC ignores polynomial factors in work
- Good parallel algorithms have polynomial depth



A brief history of parallel models



The 80s: The decade of the PRAM

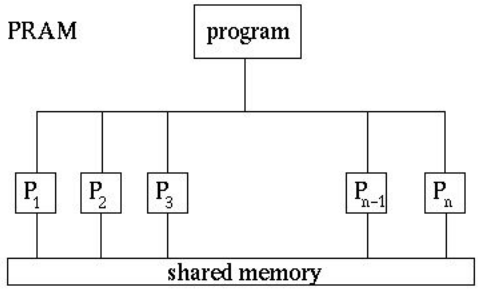
100s of papers on the topic

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

A brief history of parallel models



The 80s: The decade of the PRAM

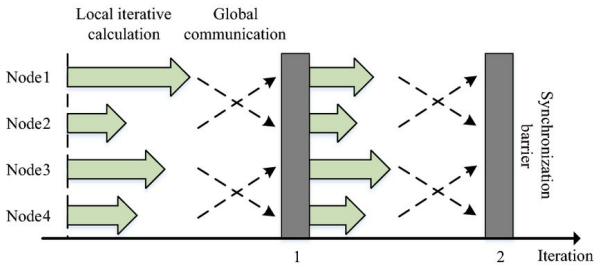
100s of papers on the topic

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

Real problem is assuming P synchronous processors

- User needs to write their own “scheduler”

A brief history of parallel models



The 90s: The PRAM gone afoul

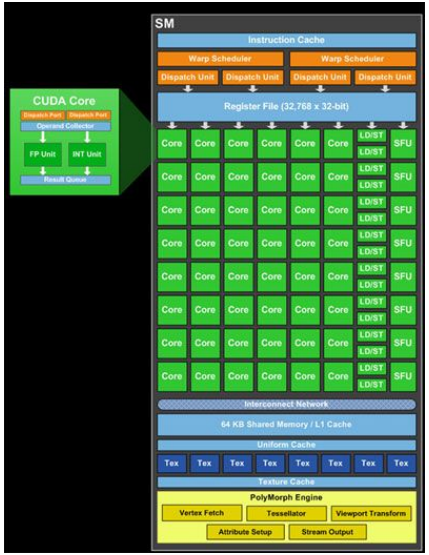
The log* failure: focus on non-robust details of the PRAM

Various more “realistic” models:

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

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

A brief history of parallel models



The 00s+: special purpose

GPU Models:

- To many details for a general model

Map-reduce models, e.g. MPC:

- Bulk synchronous is limiting
- Not show to lead to efficient algorithm design

Domain specific models

- Not general

Based on this experience, what would it take?

- High level of abstraction
- Simple code in familiar languages
- Nested parallelism
- Adaptable to account for locality
- More use of collection-based operations
- More use of higher-order functions (and lambda's)
- Adaptable to account for locality
- Architecture buy in
- **Faculty buy in (the hardest part)**
- Other suggestions: ???

Claim: we have a solution, at least in a bounded context (modulo “faculty buy in”)

- i.e. shared memory, or shared address space multicore machines

Thread based fork-join, ie. work-depth, models (binary forking model)

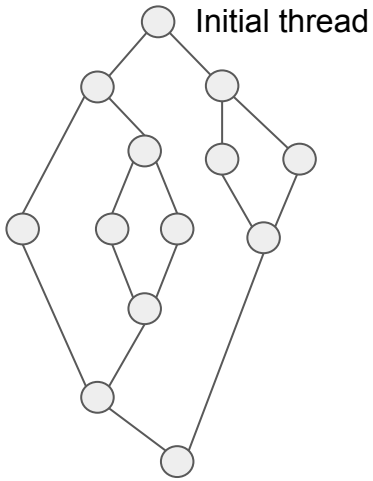
The binary forking model

Computation model:

- A set of threads each acting like a RAM on a shared memory
- One initial thread.
- A fork instruction creates two identical child threads
- When both children finish, the parent continues

Memory consistency:

- Nondeterministic total order of operations consistent with partial order defined by the fork-joins.
- Typically use race-free algorithms, which are deterministic
- Can include test-and-set and compare-and-swap.



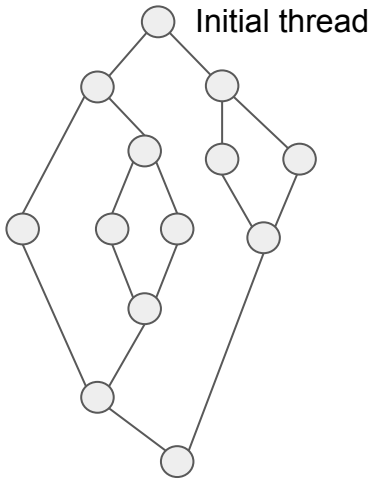
The binary forking model

Computation model:

- A set of threads each acting like a RAM on a shared memory
- One initial thread.
- A fork instruction creates two identical child threads
- When both children finish, the parent continues

Cost model:

- Work = total number of instructions
- Span = longest dependence chain
- Parallelism = Work/Span (approx # of processors can utilize)



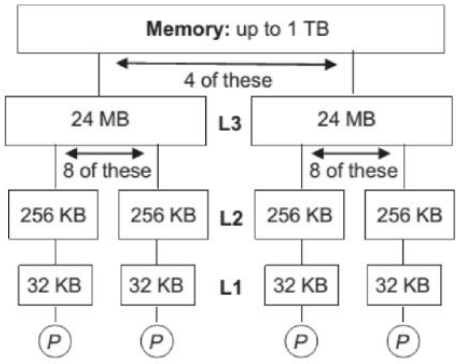
On top of this, basic collection operations

Map, filter, reduce, scan, group_by, count_by, flatten, find_first, tabulate

Most can be implemented in O(n) work and O(log n) span.

Many of these are already libraries routines in Python, C++, Java, etc (although not necessarily parallel)

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

Is binary forking a good model?

Based on this over the past 10 years we have:

Implemented a basic libraries of primitive operations (parlay, ligra, pam)

Over 50 algorithms in a wide variety of fields, for almost all

- Use ideas from algorithms community (e.g. PRAM)
- Show asymptotic bounds
- Run compared to best sequential algorithms
- Run compared to best other parallel algorithms

Used it in our intro data structures course for almost 10 years

Algorithms (implementation + cost analysis)

Sorting and searching [SPAA10, SPAA16, PPOPP18]:

- (8) Quicksort, merge, mergesort, sample sort, integer sort, hash map, binary search trees

Graph algorithms [PPOPP13,SPAA14,SODA15,SPAA17,SPAA18,VLDB20]:

- (20+) BFS, shortest paths, connectivity, spanning forest, biconnectivity, strongly connected components, minimum spanning forest, maximal independent set, maximal matching, graph coloring, k-core, approximate densest sugraph, triangle counting, widest path, betweenness centrality, spanners, low-diameter decomposition, pagerank

Algorithms (implementation + cost analysis)

String algorithms [JDA17,TOPC14] :

- (9) suffix arrays, suffix trees, wavelet tree, word count, BW, invertex index, LCP, knuth-morris-pratt hashing, huffman coding

Geometry algorithms [SPAA20, JACM20, Alenex19]:

- (7) Delaunay triangulation, convex hull, mesh refinement, k-nearest neighbor, 2d range search, 2d line intersection, 2d rectangle intersection

Database queries [VLDB20]

- (22) All 22 of the TPCH benchmarks

All are fastest or close to fastest on shared memory machines.

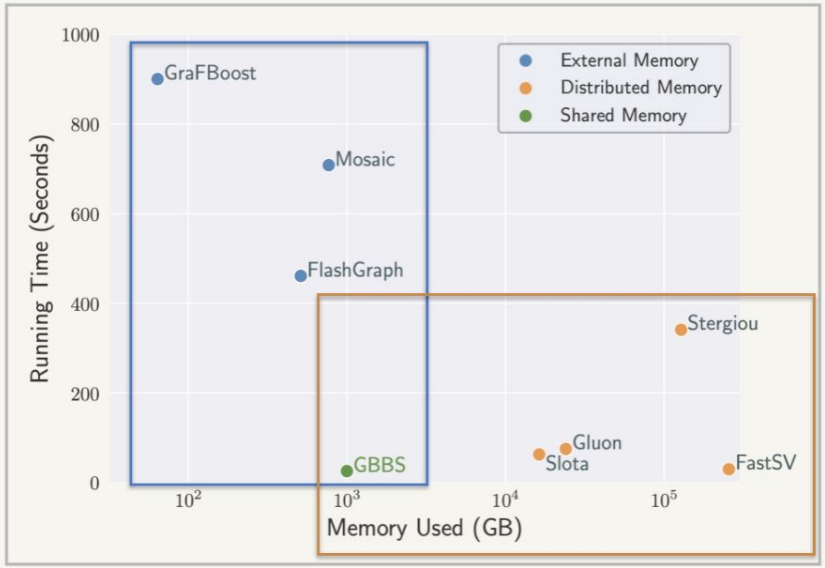
All have simple theoretical bounds in terms of work and span (not all are polylogarithmic span, and some depend on input characteristics, e.g. graph diameter)

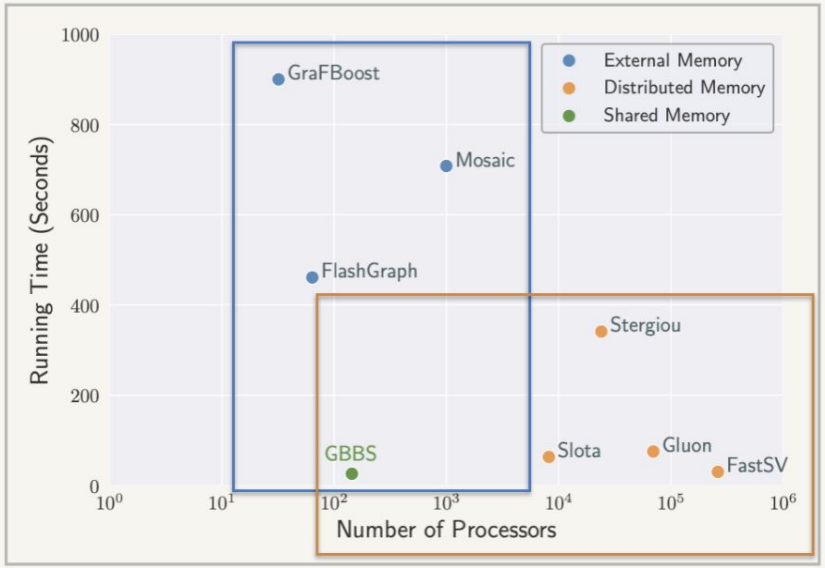
In many cases also faster than much larger distributed memory machines.

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	$O(n+m)$	$O(\log n)$

Benchmarking Connectivity on WebDataCommons Graph





Outperform external memory results by orders of magnitude using comparable hardware.

Outperform distributed memory results using orders of magnitude less hardware.

But is the code simple?

Some examples:

- Quicksort
- BFS
- Graph connectivity
- Merging

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 <, =, >

Breadth First Search

```
vSequence BFS(vertex start, const Graph &G) {
  size_t n = G.numVertices();
  vSequence parent(n, -1);
  parent[start] = start;
  auto frontier = ligra::vertex_subset(start);

  while (frontier.size() > 0)
        frontier = ligra::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

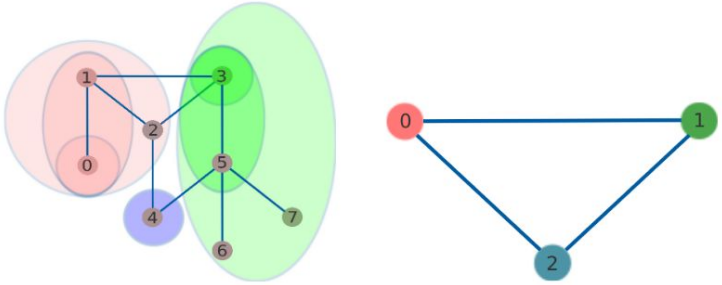
Maps over out-edges of each vertex in the frontier

Graph Connectivity

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

```
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;
}
```

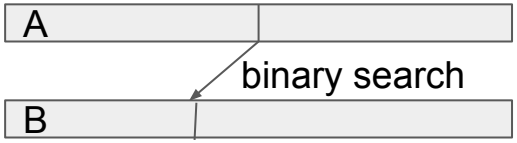


(a) graph decomposition (b) contracted graph

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);}};
  }}
```



Summary:

- Simple code
- Not very different from sequential algorithms
- Common techniques (D&C, contraction, …)
- Easy analysis (a bit more than for seq algorithms)
- Can analyze for locality (spatial + temporal) with same code
- Leads to fast code
- Supported by existing machines

Caveats : not directly relevant to all parallel machines

Purpose of asymptotic (big-O) analysis:

+ yes

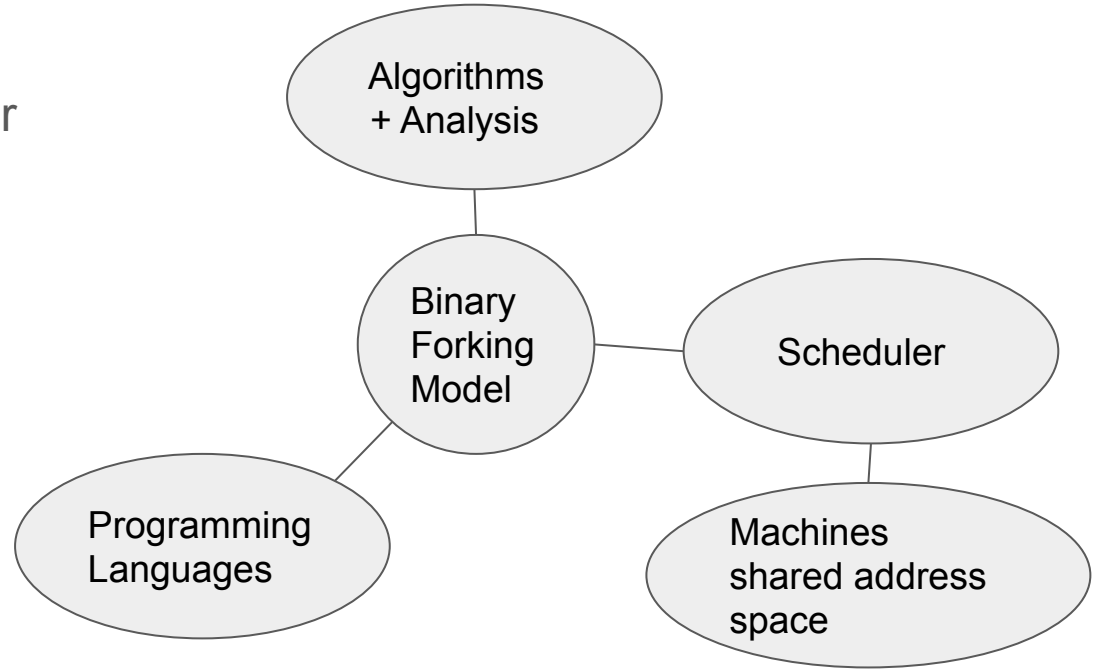
1. **Abstraction** : avoid details
2. **Guidance** : towards a good algorithm
3. **Scalability** : how will cost grow with size
4. **Justify** : algorithms, data structures and techniques

- 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)

Can we get the same “ecosystem” for parallelism

The binary-forking model
A great bridging model, at least for some class of machines.



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

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

```
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);
}
```

Declares whitespace