Parallel Algorithms and Big Data Für Alle

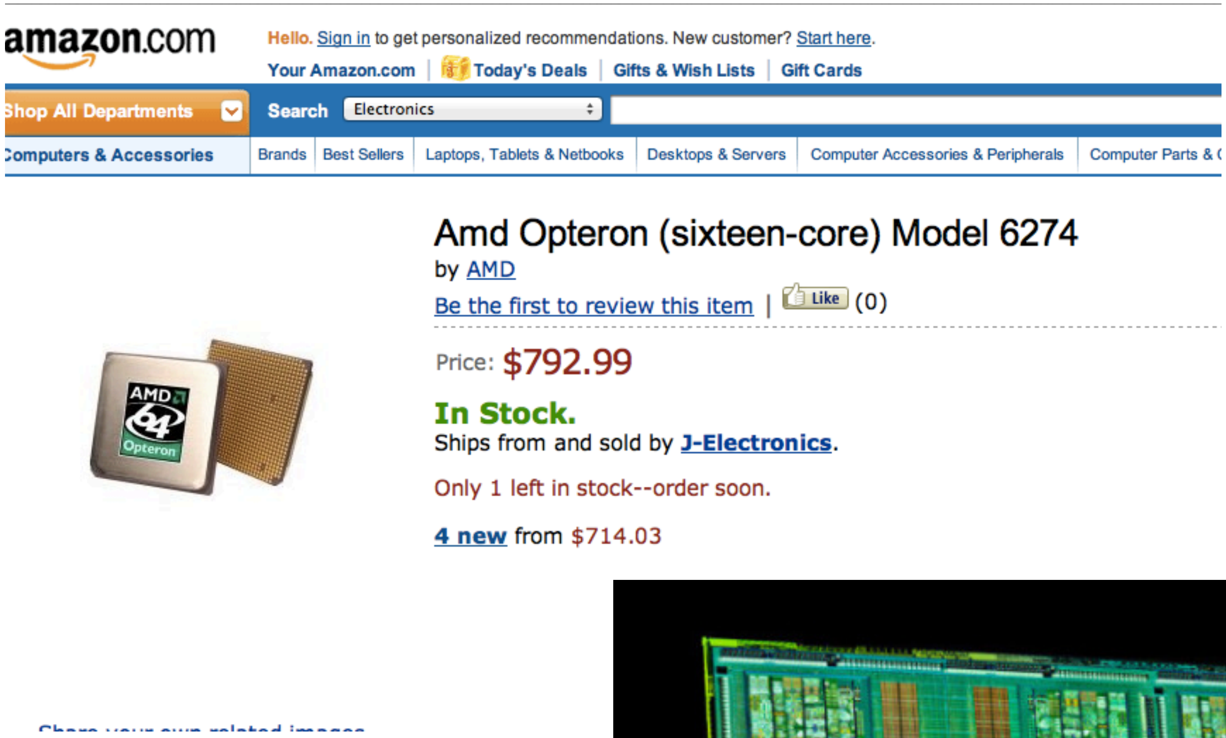
★

Guy Blelloch,

and lots of others

Carnegie Mellon University

Why Parallelism?



amazon.com

Hello. Sign in to get personalized recommendations. New customer? Start here.

Amd Opteron (sixteen-core) Model 6274

by AMD

Be the first to review this item | Like (0)

Price: $792.99

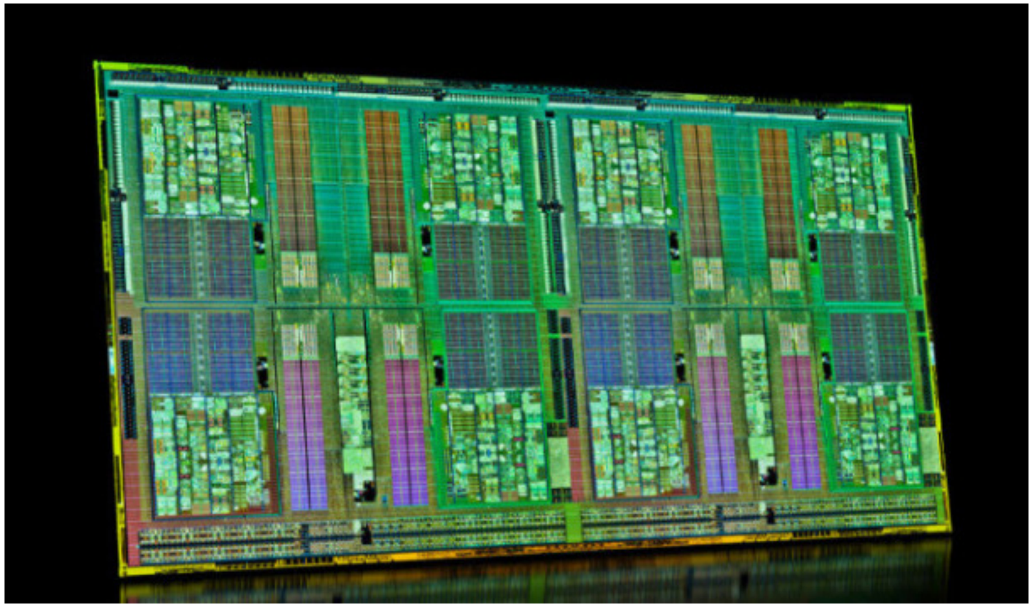
In Stock.

Ships from and sold by J-Electronics.

Only 1 left in stock--order soon.

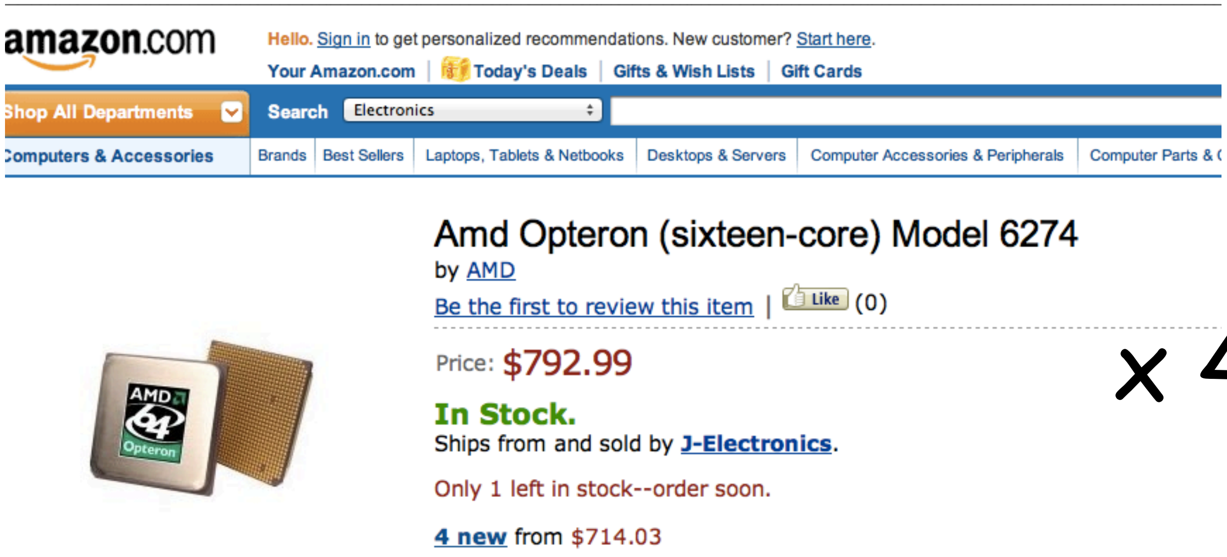
4 new from $714.03

Share your own related images



IC 2015

64 core blade servers ($6K) (shared memory)


amazon.com
Hello. Sign in to get personalized recommendations. New customer? Start here.
Your Amazon.com | Today's Deals | Gifts & Wish Lists | Gift Cards
Shop All Departments
Search
Electronics
Computers & Accessories
Brands
Best Sellers
Laptops, Tablets & Netbooks
Desktops & Servers
Computer Accessories & Peripherals
Computer Parts & C

Amd Opteron (sixteen-core) Model 6274
by AMD
Be the first to review this item | Like (0)

Price: $792.99

In Stock.
Ships from and sold by J-Electronics.

Only 1 left in stock--order soon.

4 new from $714.03

x 4 =

4992 “cuda” cores

Roll over image to zoom in

Nvidia Tesla K80 24GB GPU Accelerator passive cooling 2x Kepler GK210 900-22080-0000-000

by NVIDIA

★★★★☆ 29 customer reviews | 11 answered questions

Price: $4,295.95 + $11.55 shipping

Note: Not eligible for Amazon Prime.

In Stock.

Ships from and sold by eServer PRO.

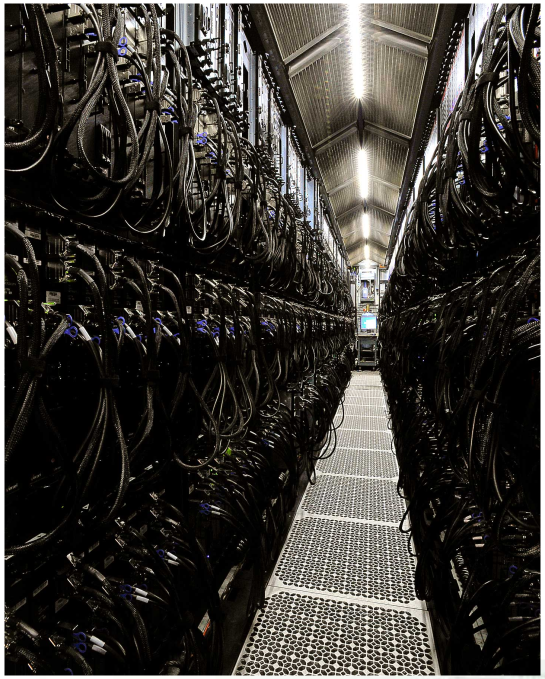
Estimated Delivery Date: Aug. 27 - Sept. 1 when you choose Expedited at checkout.

- Nvidia Tesla K80 GPU: 2x Kepler GK210
- Memory size (GDDR5) : 24GB (12GB per GPU)
- CUDA cores: 4992 (2496 per GPU)
- Memory bandwidth: 480 GB/sec (240 GB/sec per GPU)
- 2.91 Tflops double precision performance with NVIDIA GPU Boost - See more at: http://www.nvidia.com/object/tesla-servers.html#sthash.IF5LVwFq.dpuf

4 new from $4,135.00

Upgrading to a Solid-State Drive?

Learn how to install an SSD with Amazon Tech Shorts. Learn more



Up to 300K servers

Samsung Galaxy S IV to feature Exynos 28nm quad-core processor?

Written by Andre Yoskowitz @ 01 Nov 2012 18:02

It has been a few weeks but there is a new rumor regarding the upcoming Samsung Galaxy S IV.

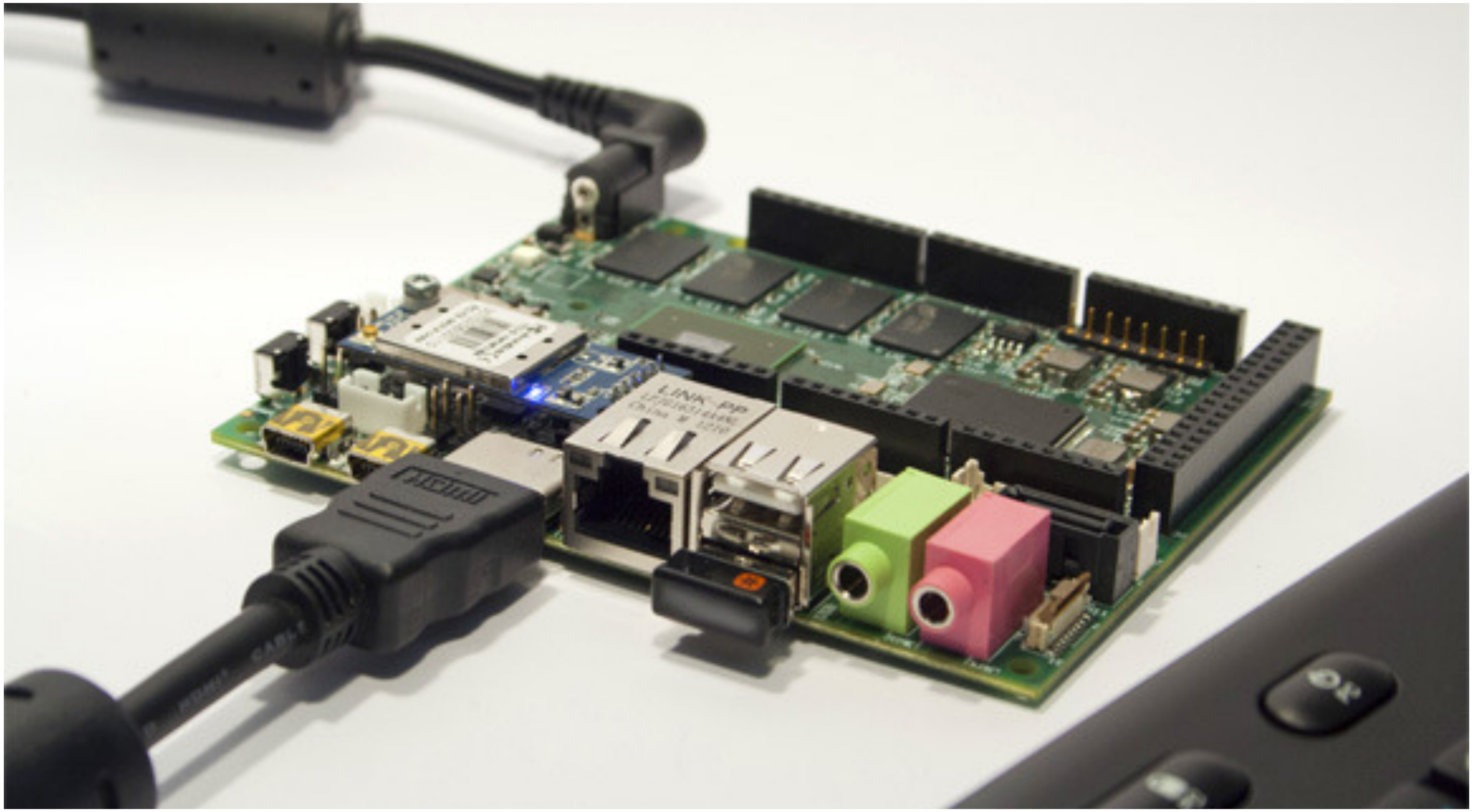
According to reports, Samsung will pack next year's flagship device with its "Adonis" Exynos processor, a quad-core ARM 15 beast that uses efficient 28nm tech.

Samsung is supposedly still testing the application processor, but mass production is scheduled for the Q1 2013 barring any delays.

Forget Quad-Core: Intel Working on 48-Core Smartphone and Tablet Processors

By Todd Haselton on October 31, 2012 in Hardware

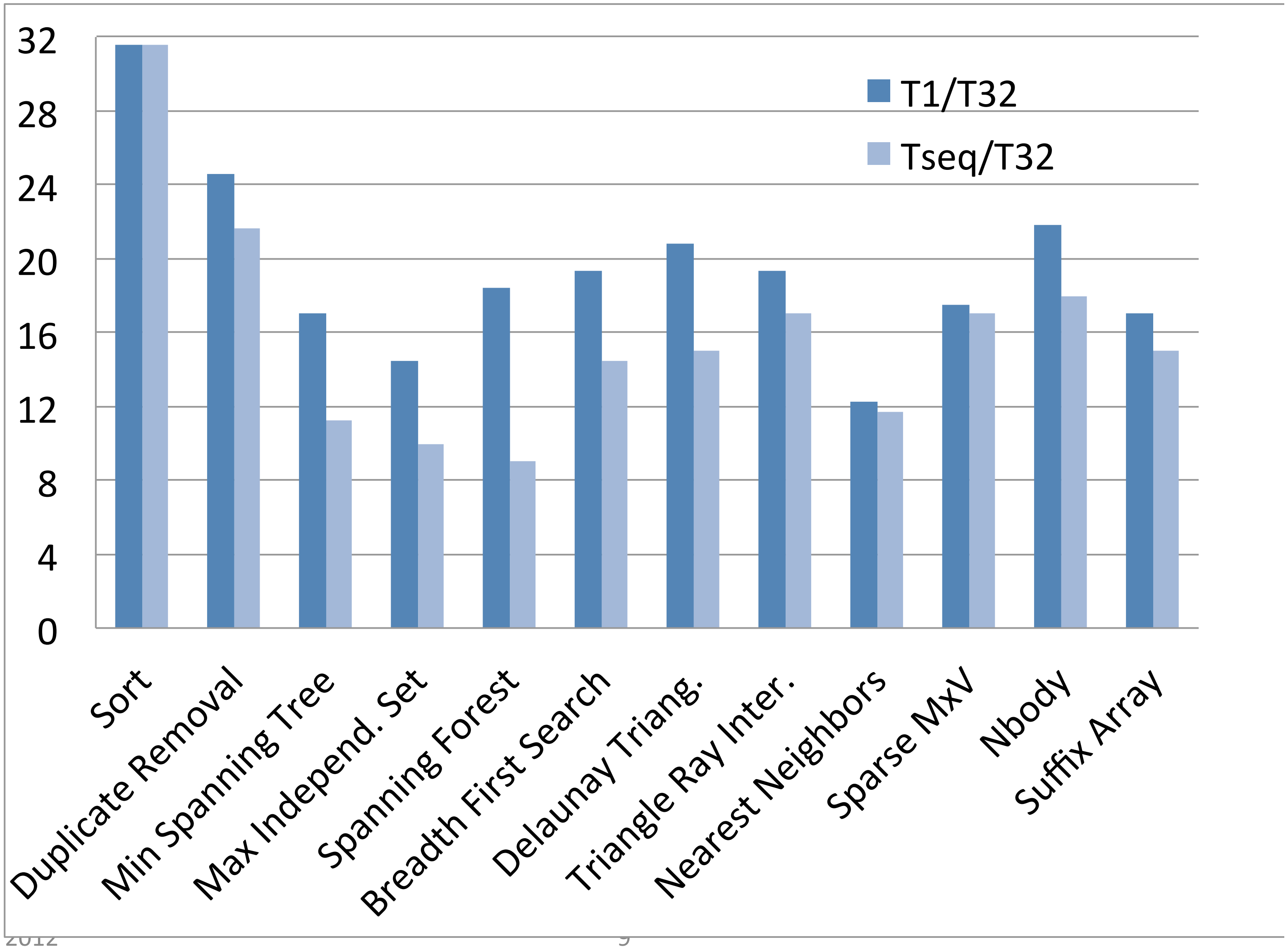
UDOO : Quad Core



Parallel machines have replaced sequential machines, but parallel algorithms have not yet replaced sequential algorithms. Why?

It is not because they are not efficient or cost effective.

32 Cores off the shelf machine



Parallel machines have replaced sequential machines, but parallel algorithms have not yet replaced sequential algorithms. Why?

More likely because Parallel Algorithms are <u>**viewed**</u> as **hard**, **messy**, and **theory does not match practice**.

Why are Sequential Algorithms so Successful?

- exactly predi❌runtimes?
- are good for highly❌ning optimized codes?
- will impress our friends? Maybe

Why are Sequential Algorithms so Successful?

1. Well defined and simple cost model which is “good enough” for asymptotic comparisons
2. Simple pseudocode and small step to real code that can be easily compiled and run to get reasonably efficient code.
3. Good for explaining core ideas, and why they are useful
4. Sequential algorithms are elegant

Quicksort (AHU78)

```
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
```

My Focus

Parallel algorithms should be equally elegant, simple, efficient in practice, and efficient in theory.

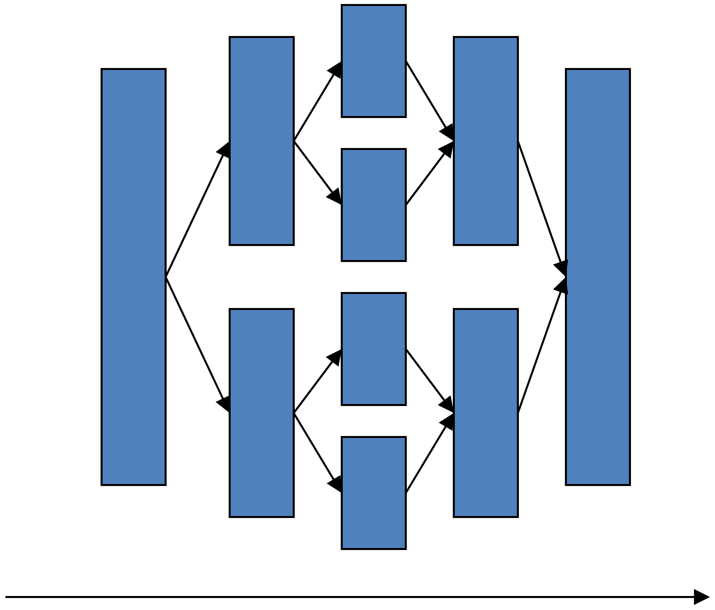
- Our core algorithms-complexity course at CMU taught to all Sophomores now uses parallelism from the start.

Quicksort (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];
```

Quicksort (nested parallelism)

- Analyze in terms or Work (W) and Depth (D)



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

Work = $O(n \lg n)$

Parallelism = $W/D = O(n/\lg n)$

Time = $W/P + D$
P = # processors

Rest of Talk

- Sequential Iterative Algorithms
- Ligra: A graph processing framework

Sequential Iterative Algorithms

```
for i from 1 to n
    do something;
```

Work from SPAA13, SODA15

Sequential Iterative Algorithms

```
for i from 1 to n
    a[i] = b[i] + 1;
```

Is this parallel?

```
parallelFor i from 1 to n
    a[i] = b[i] + 1;
```

Sequential Iterative Algorithms

```
for i from 1 to n
    swap(A[rand(i)],A[i])
```

Is this parallel?

Sequential Iterative Algorithms

★
```
for i from 1 to n
    swap(A[rand(i)],A[i])
```

```
for i from 1 to n
    SearchTreeInsert(T,A[i])
```

★
```
S[1..n] = 0
for i from 1 to n
    if for all u in N(V[i]), S[u]=0
    then S[v] = 1
```

Sequential Iterative Algorithms

```
for i from 1 to m
    u = F.find(E[i].u)
    v = F.find(E[i].v)
    if (u != v) F.union(u,v)
```

Others:

- List contraction
- Tree contraction
- Maximal Matching

Sequential Iterative Algorithms

Why do we care if parallel?

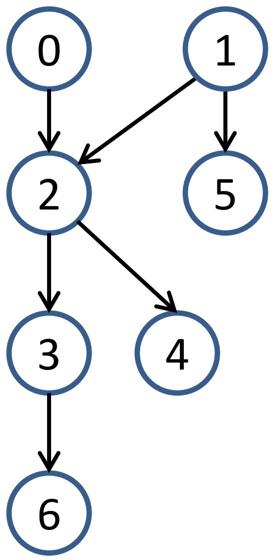
- Simple parallel code
- Perhaps fast algorithms
- Intellectual curiosity
- Determinism

How do we analyze?

Iteration Dependence Graph

Sequential iterative algorithm

```
for i in {0,…,n-1}
    do something;
```

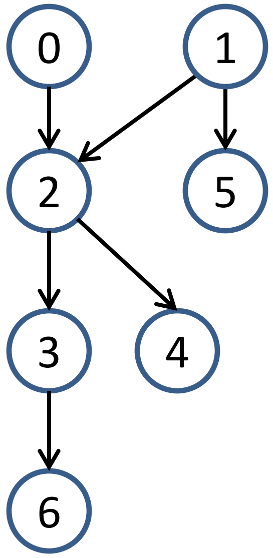

$$\begin{tikzcd}
0 \arrow[d] & 1 \arrow[dl] \arrow[d] \\
2 \arrow[d] \arrow[dr] & 5 \\
3 \arrow[d] & 4 \\
6 &
\end{tikzcd}$$

- Each iterate is a vertex
- i → j means iterate i must execute before iterate j
- Can execute in parallel if respecting dependencies
- Graph is dependent on input data

Iteration Dependence Graph

Sequential iterative algorithm

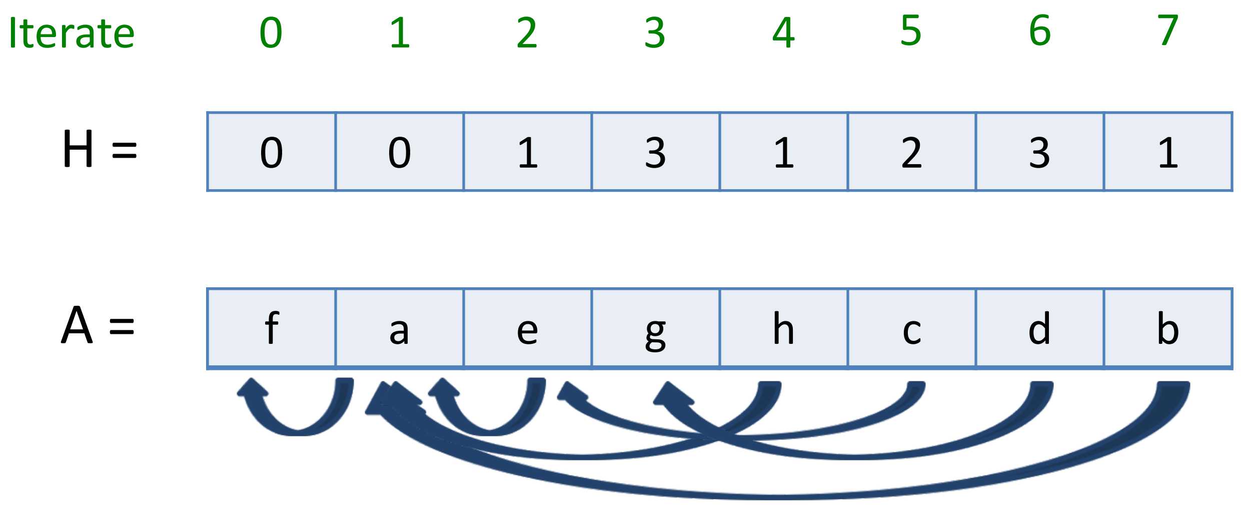
```
for i from 1 to n
    do something;
```



1. what is depth of the graph?
2. can we easily detect dependences?

Random Permutation [Durstenfeld, Knuth]

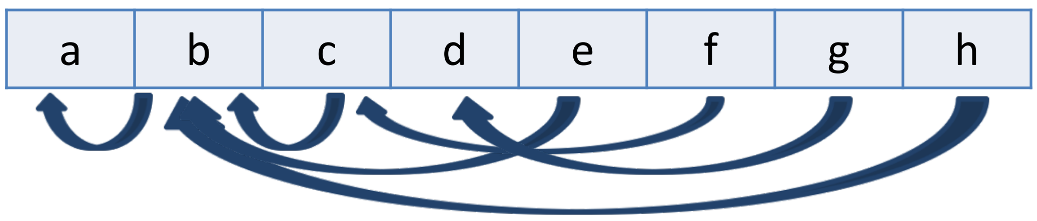
```
for i from n to 1
    H[i] = rand(i)
for i from n to 1
     swap(A[H(i)],A[i])
```



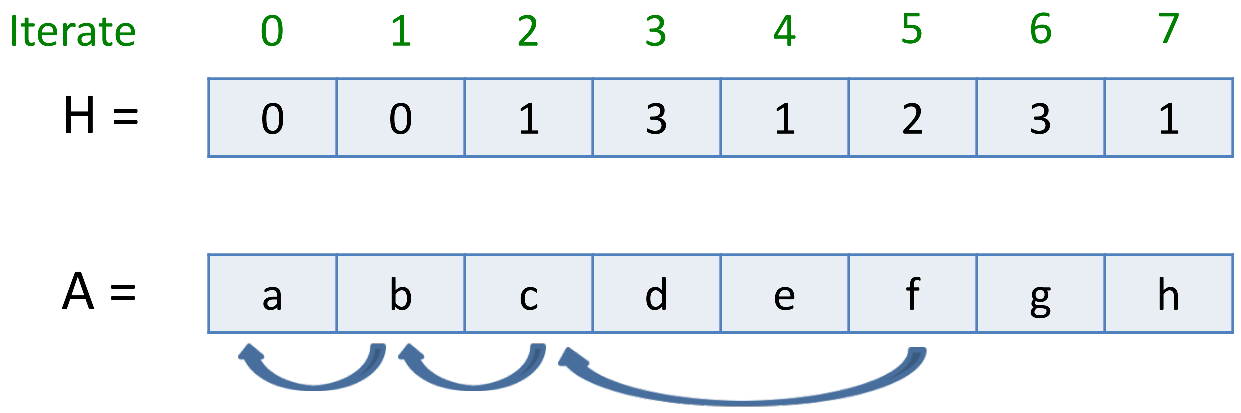
Iterate	0	1	2	3	4	5	6	7
H =	0	0	1	3	1	2	3	1
A =	f	a	e	g	h	c	d	b

Is this parallel?

Iterate	0	1	2	3	4	5	6	7
H =	0	0	1	3	1	2	3	1
A =	a	b	c	d	e	f	g	h



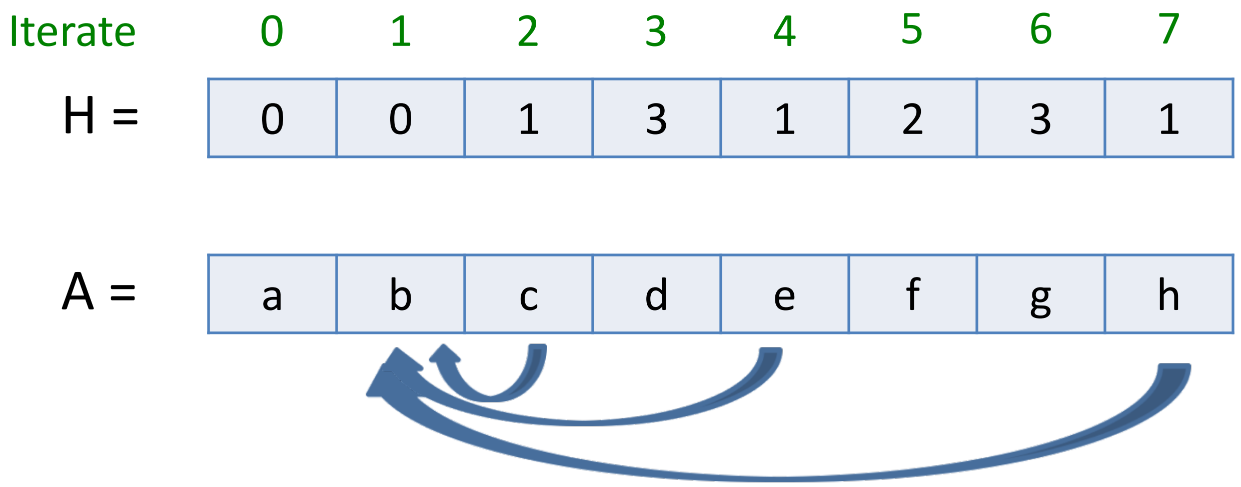
Is this parallel?



Iterate	0	1	2	3	4	5	6	7
H =	0	0	1	3	1	2	3	1
A =	a	b	c	d	e	f	g	h

- “Swap chains” have sequential dependence

Is this parallel?

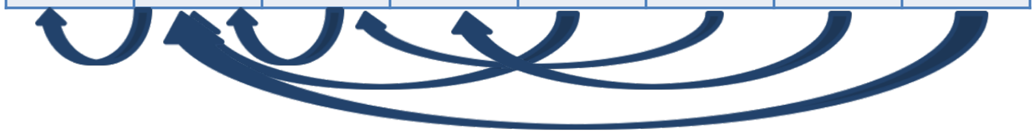


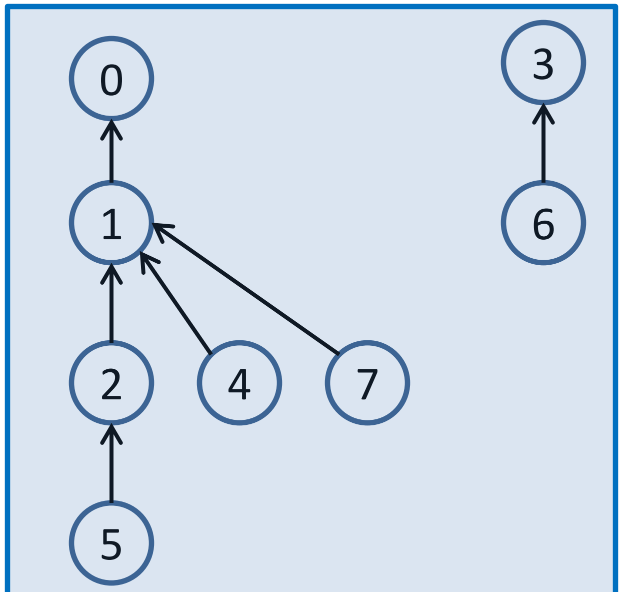
Iterate	0	1	2	3	4	5	6	7
H =	0	0	1	3	1	2	3	1
A =	a	b	c	d	e	f	g	h

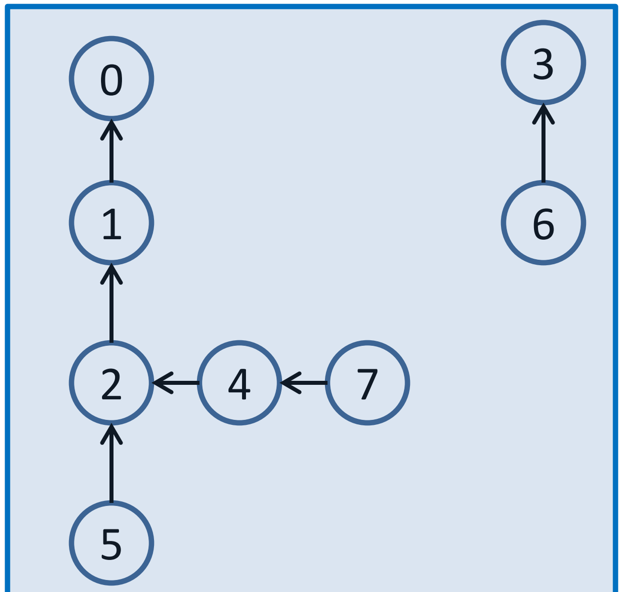
- “Swap chains” have sequential dependence
- Each location that is the target of multiple swaps has sequential dependence
- Can execute multiple iterates in parallel as long as dependencies are respected

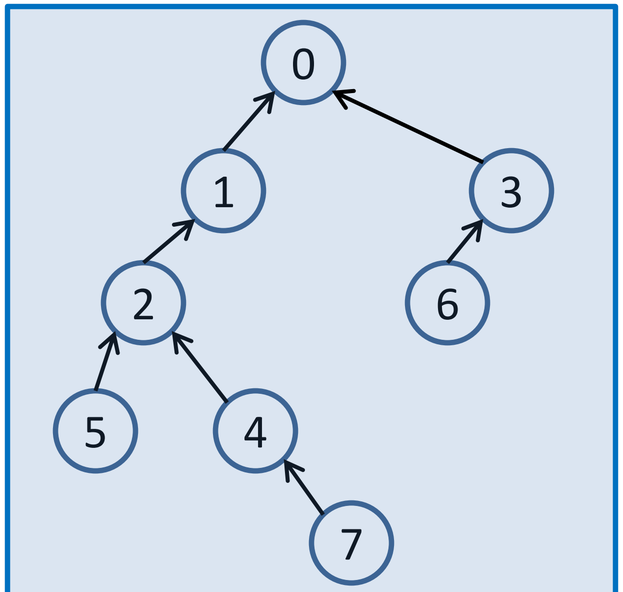
Random Permutation Iteration Depth

Iterate	0	1	2	3	4	5	6	7
H =	0	0	1	3	1	2	3	1
A =	a	b	c	d	e	f	g	h



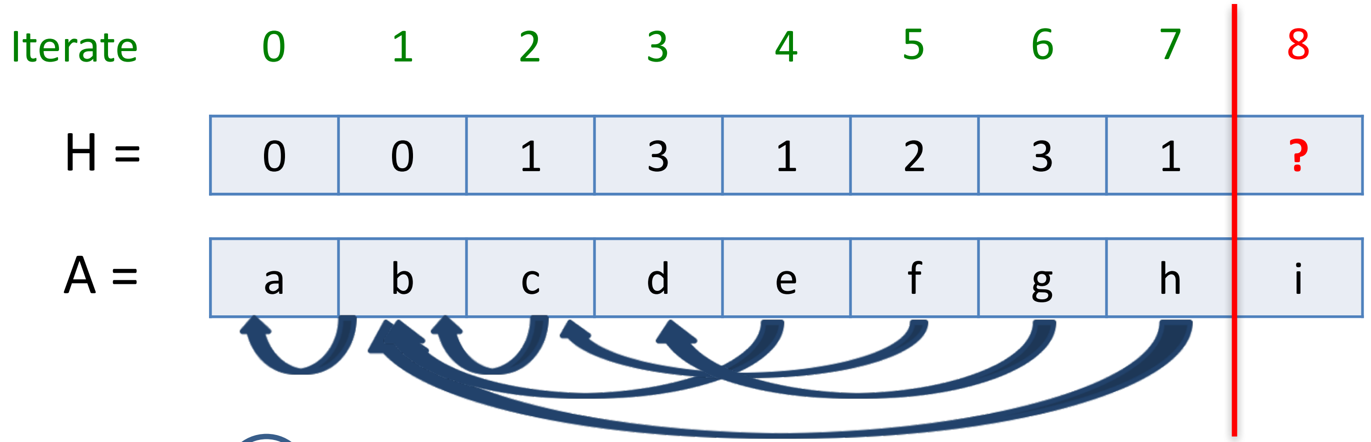

Dominance Forest

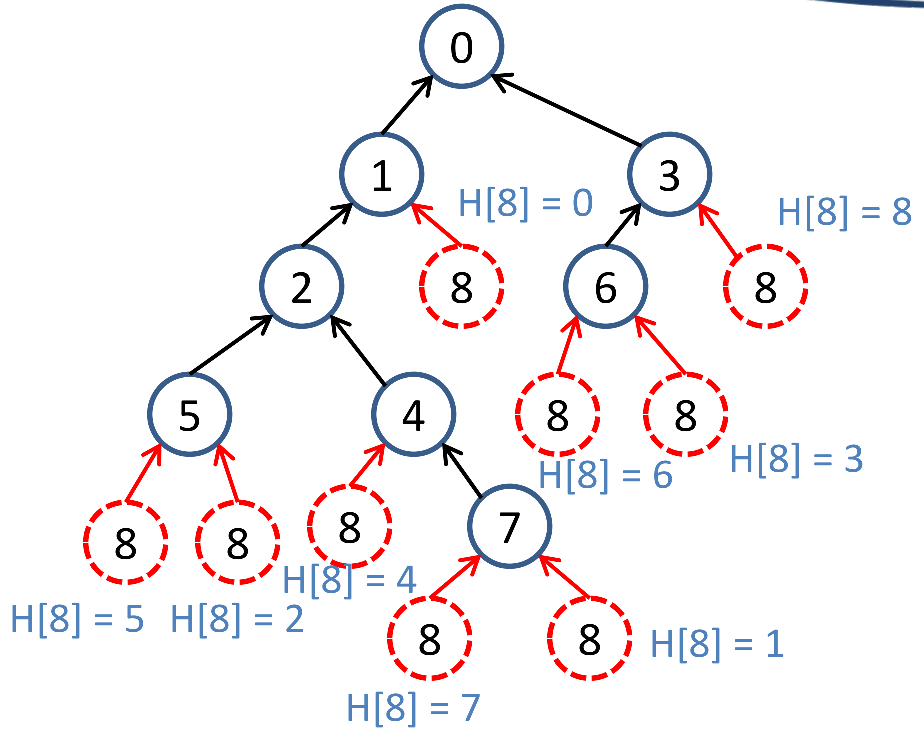

Dependence Forest


Linked Dependence Tree

Random Permutation Iteration Depth

Iterate	0	1	2	3	4	5	6	7	8
H =	0	0	1	3	1	2	3	1	?
A =	a	b	c	d	e	f	g	h	i





- Each value of H[8] corresponds to a unique location in binary tree
- All possible locations equally likely
- Corresponds to construction of a random binary search tree!

Iteration Depth

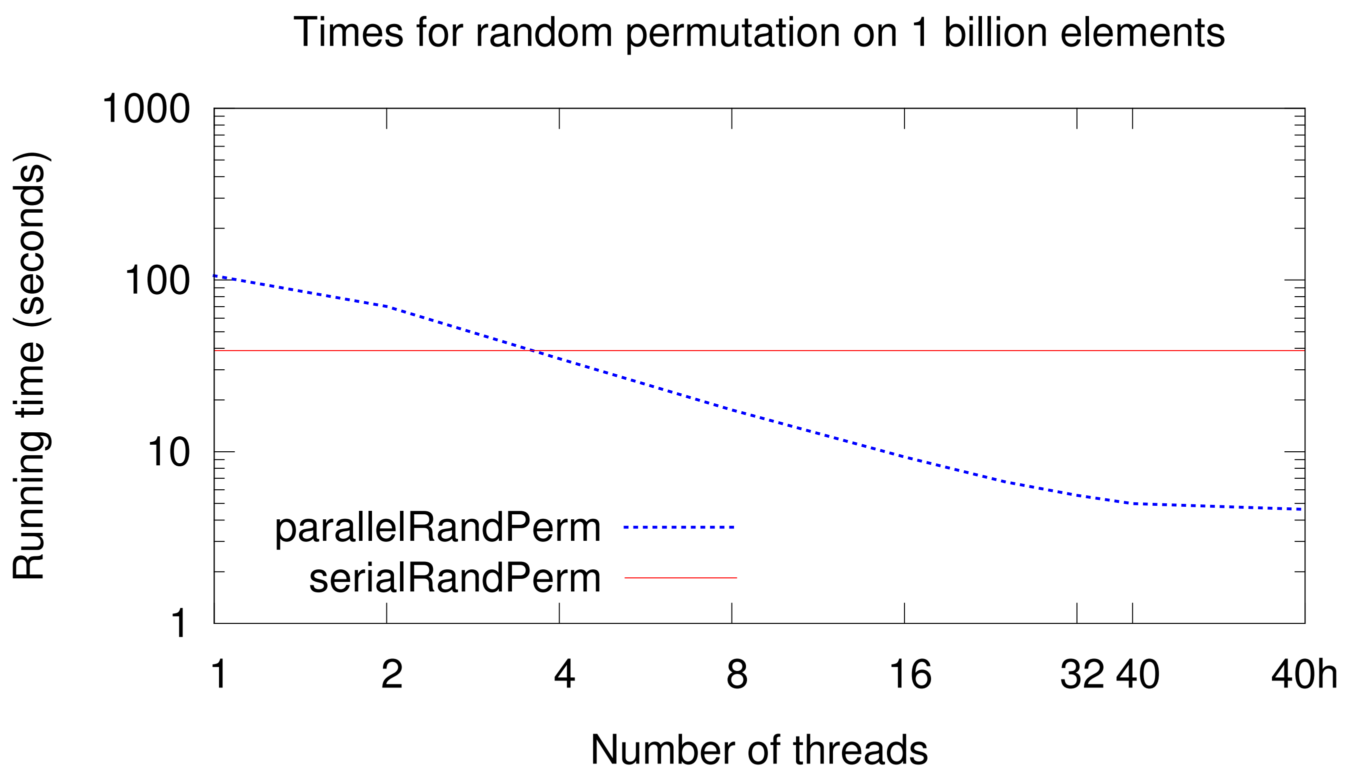
- Height of a random binary search tree on n nodes is θ(log n) w.h.p. [Devroye '86]
- Therefore, iteration depth of random permutation is O(log n) w.h.p.
- Can also show that linear work, even if every node tries on every step

- Not best: O(log* n) depth w.h.p. [Hagerup '91]

Detecting Dependences

```
for i from 1 to n
    H[i] = rand(i)
parallelFor i from 1 to n
    R[H(i)] = i; R[i] = i;        ← Priority write
    if R[H(i)] == i and R[i] == i
    then swap(A[H(i)],A[i])
    else “try again”
```

Performance



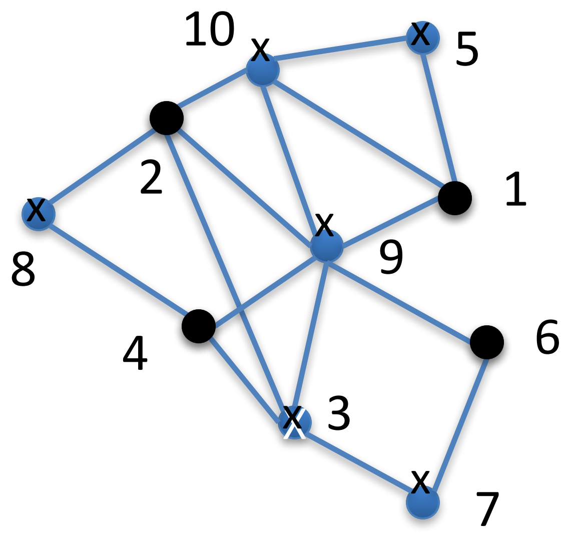
3x slower on 1 core

9x faster on 40 cores

Maximal Independent Set

Sequential algorithm:

```
for i in 1 to n : S[i] = Undecided
for i in 1 to n
    if for all j in N(V[i]), v < u, S[j] = Out
    then S[j] = In
    else S[j] = Out
```



Maximal Independent Set

Sequential algorithm:

```
for i in 1 to n : S[i] = Undecided
for i in 1 to n
    if for all j in N(V[i]), v < u, S[j] = Out
    then S[j] = In
    else S[j] = Out
```

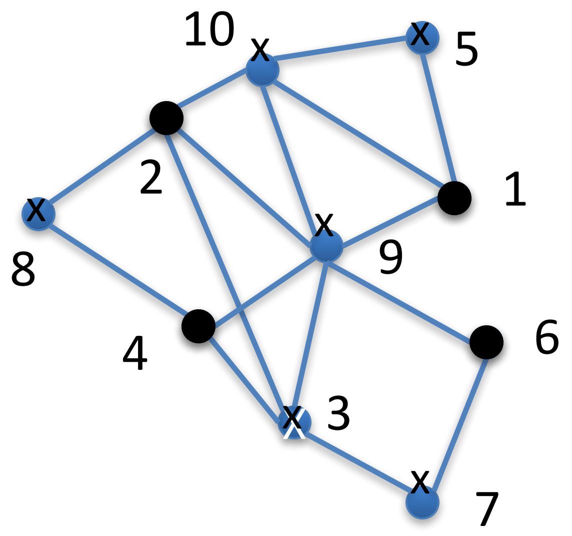
Very efficient: most edges not even visited, simple loops

About 7x faster than sorting m edges

Maximal Independent Set

Same algorithm: with parallel speculation

```
for i in 1 to n : S[i] = Undecided
for i in 1 to n
    if for all j in N(V[i]), v < u, S[j] = Out
    then S[j] = In
    else S[j] = Out
```



Iteration Depth/Performance

- For random ordering of vertices: $O(\log^2 n)$
  - Non trivial, for arbitrary degree
  - $O(\log n)$ for constant degree
- Work is $O(m)$ if using prefixes
- Dependences easy to detect.
- 12x speedup on 40 cores over sequential algorithm

MIS Parallel Code

```
struct MISStep {
  bool reserve(int i) {
    int d = V[i].degree;
    flag = IN;
    for (int j = 0; j < d; j++) {
      int ngh = V[i].Neighbors[j];
      if (ngh < i) {
        if (Fl[ngh] == IN) { flag = OUT; return 1;}
        else if (Fl[ngh] == LIVE) flag = LIVE; } }
    return 1; }

  bool commit(int i) { return (Fl[i] = flag) != LIVE;}};

void MIS(FlType* Fl, vertex* V, int n, int psize)
  speculative_for(MISStep(Fl, V), 0, n, psize);}
```

Maximal Independent Set

Costs:

- Span = $O(\log^3 n)$
  Expected case over all initial permutations
- Work = $O(m)$
  if prefix size = $O(n/d_{max})$

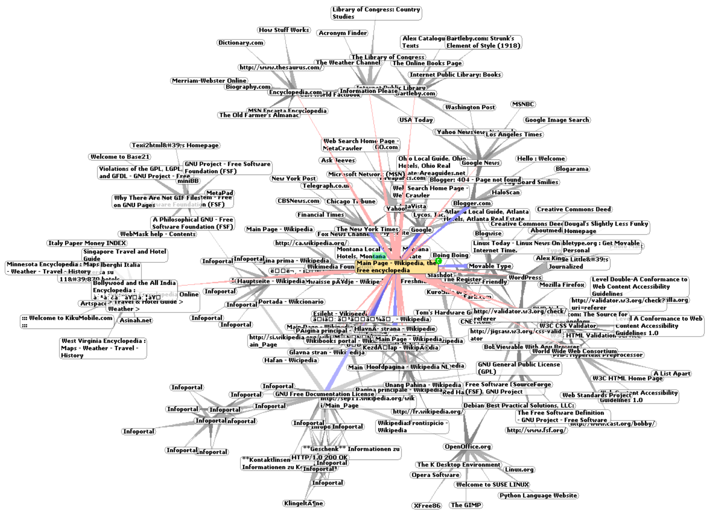
Determininistic :

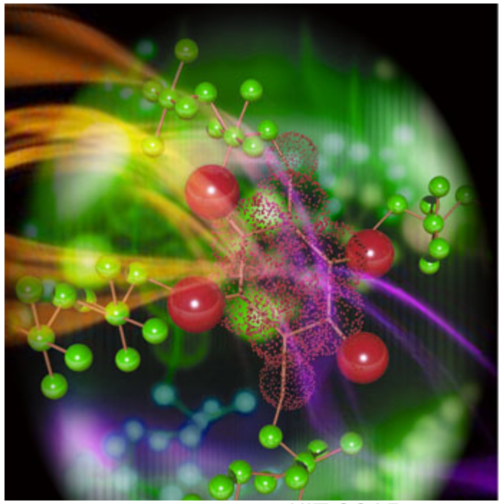
- result only depends on initial permutation of vertices

Part 2: Ligra

A Graph Processing Framework

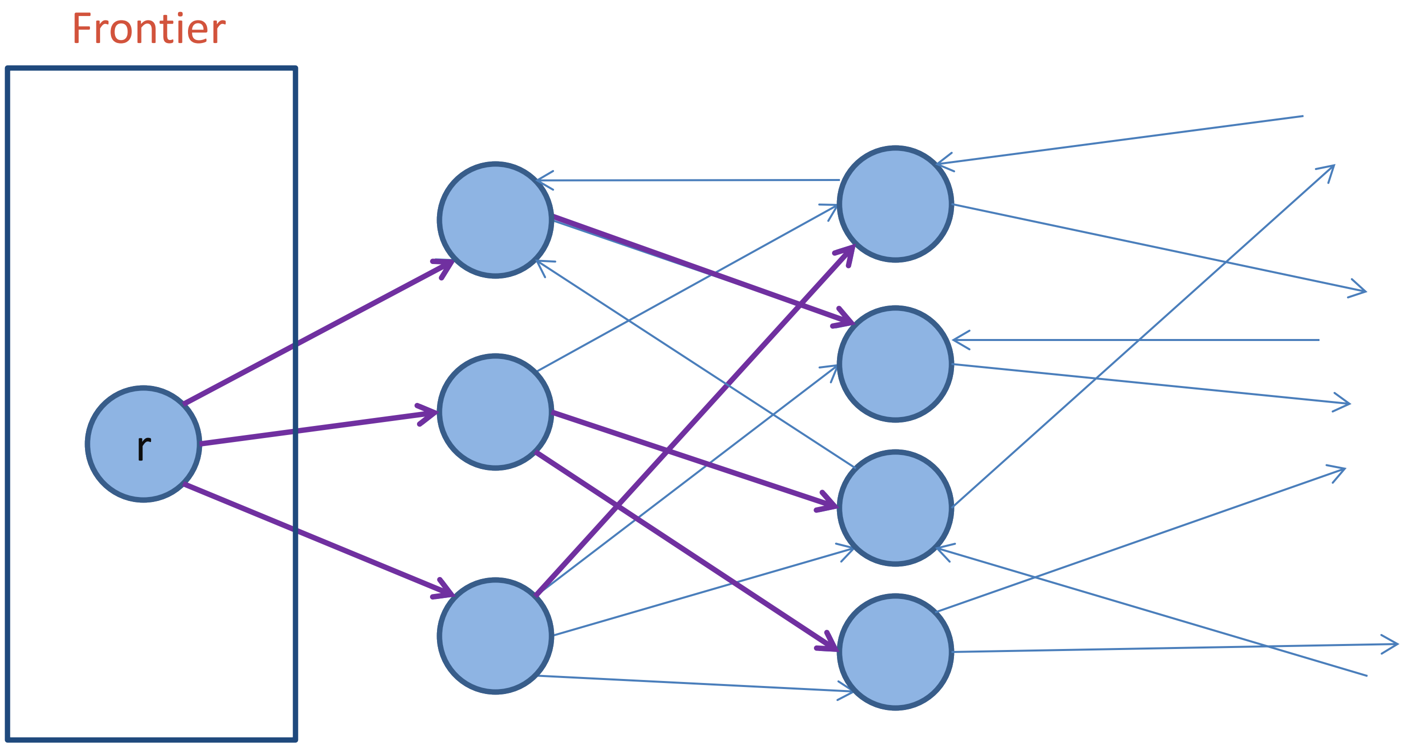
- For shared memory
- Best for frontier-based algorithms
- Space and Time efficient
- Programming efficiency
- Asymptotic bounds can be analyzed





Breadth-first Search (BFS)

- Compute a BFS tree rooted at source *r* containing all vertices reachable from *r*



- Can process each frontier in parallel
  - Race conditions, load balancing

BFS Abstractly: Frontier Based

1. Operate on a subset of vertices
2. Map computation over subset of edges in parallel
3. Return new subset of vertices
4. (Map computation over subset of vertices in parallel)

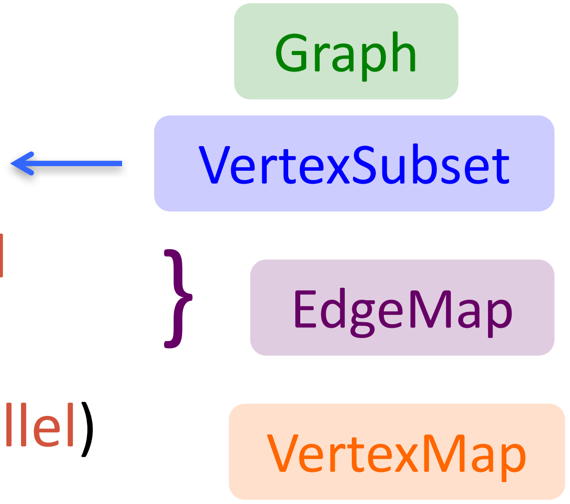
BFS visits every vertext once, but in general can visit many times. Synchronous.

Breadth-first search	Bellman-Ford shortest paths
Betweenness centrality	Graph eccentricity estimation
Connected components	PageRank
Delta stepping	Diameter estimation

Can we build an abstraction for these types of algorithms?

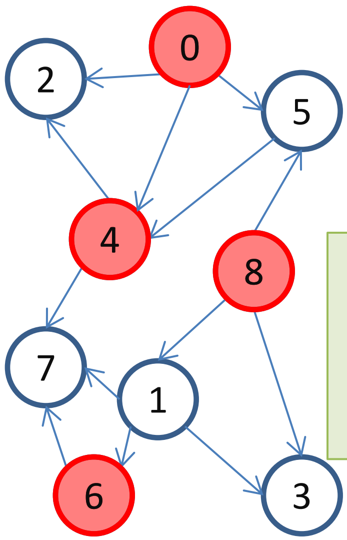
Ligra

- Operate on a subset of vertices
- Map computation over subset of edges in parallel and return new subset of vertices
- (Map computation over subset of vertices in parallel)

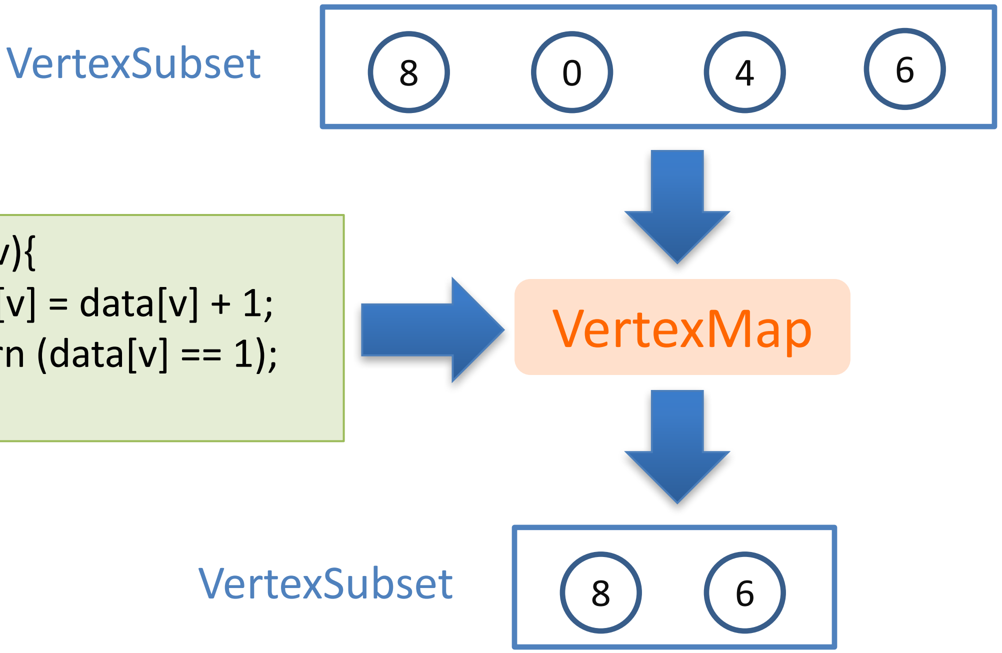


Other graph processing frameworks: Pregel/Giraph, GraphLab, Pegasus, Knowledge Discovery Toolbox, GraphChi, Parallel BGL, and many others…

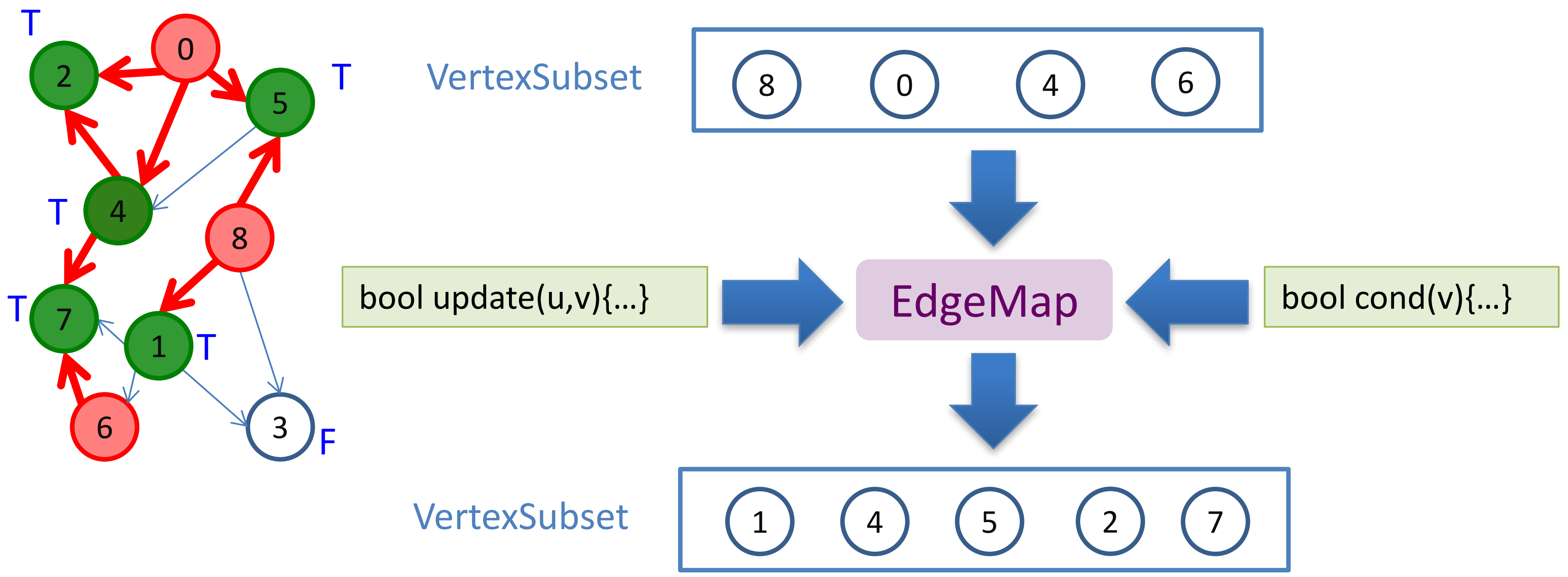
Ligra Framework



```
bool f(v){
    data[v] = data[v] + 1;
    return (data[v] == 1);
}
```



Ligra Framework



Why edge based?

- Parallel over the edges
- Sparse/dense (discussed later)

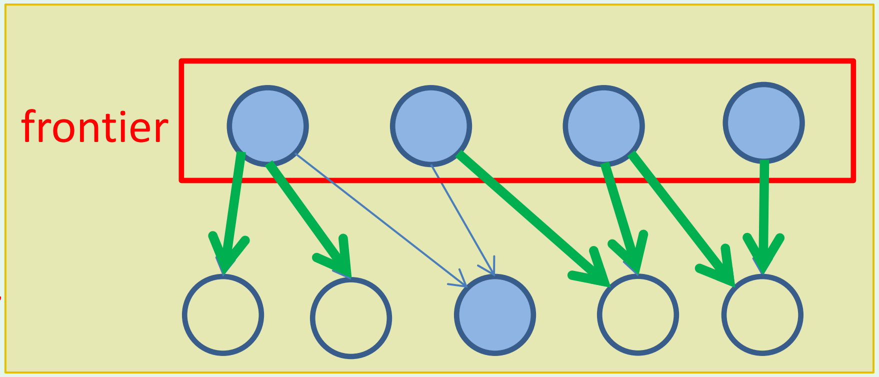
Breadth-first Search in Ligra

```
parents = {-1, …, -1};   //-1 indicates “unvisited”

procedure UPDATE(s, d):
    return compare_and_swap(parents[d], -1, s);

procedure COND(i):
    return parents[i] == -1;   //checks if “unvisited”

procedure BFS(G, r):
    parents[r] = r;
    frontier = {r}; //VertexSubset
    while (size(frontier) > 0):
        frontier = EDGEMAP(G, frontier, UPDATE, COND);
```



EdgeMap: Sparse and Dense

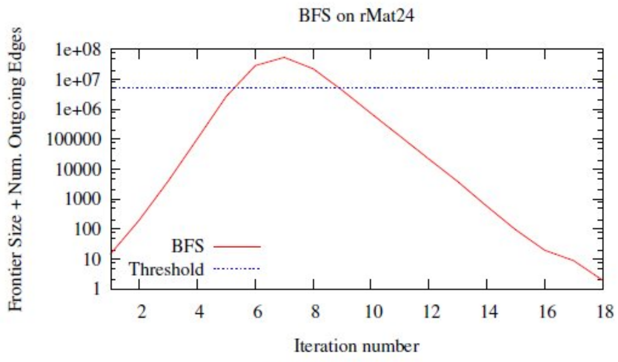
```
procedure EDGEMAP(G, frontier, Update, Cond):
    if (|frontier| + sum of out-degrees > threshold) then:
        return EDGEMAP_DENSE(G, frontier, Update, Cond);
    else:
        return EDGEMAP_SPARSE(G, frontier, Update, Cond);
```

Loop through outgoing edges of frontier vertices in parallel

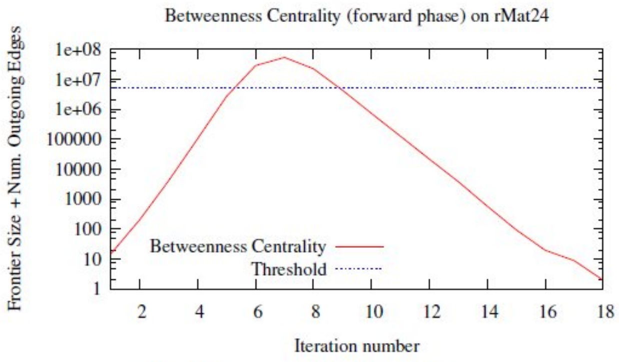
Loop through incoming edges of “unexplored” vertices (in parallel), breaking early if possible

- First used by Beemer for BFS, but Ligra shows that useful for a wide variety of algorithms

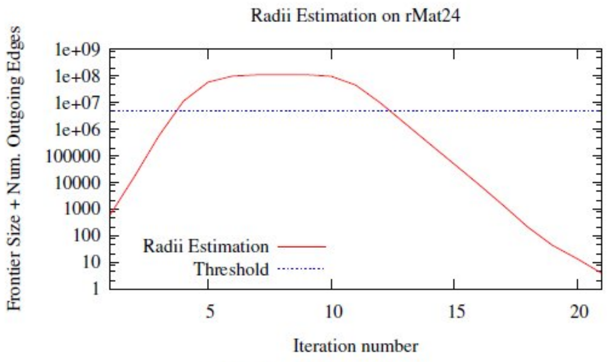
Frontier Plots



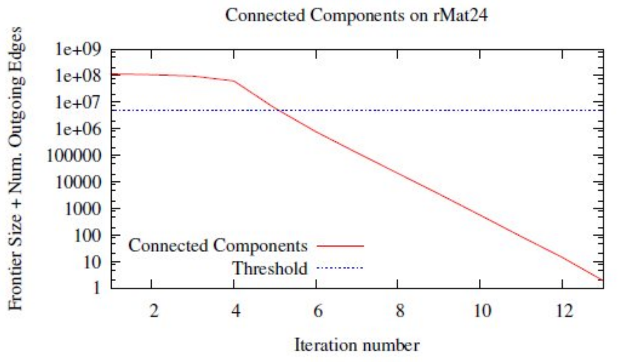
(a) BFS



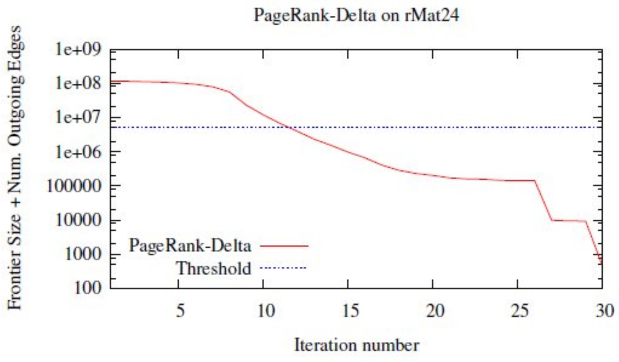
(b) Betweenness Centrality



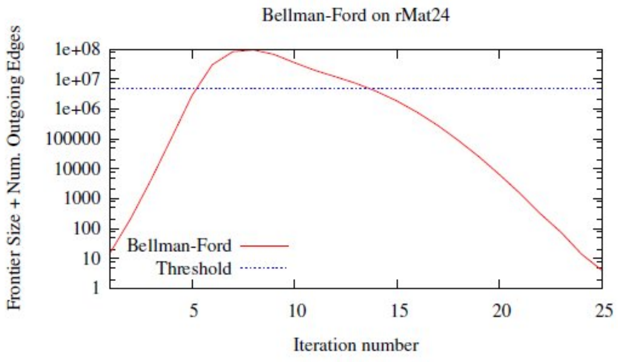
(c) Radii Estimation



(d) Connected Components

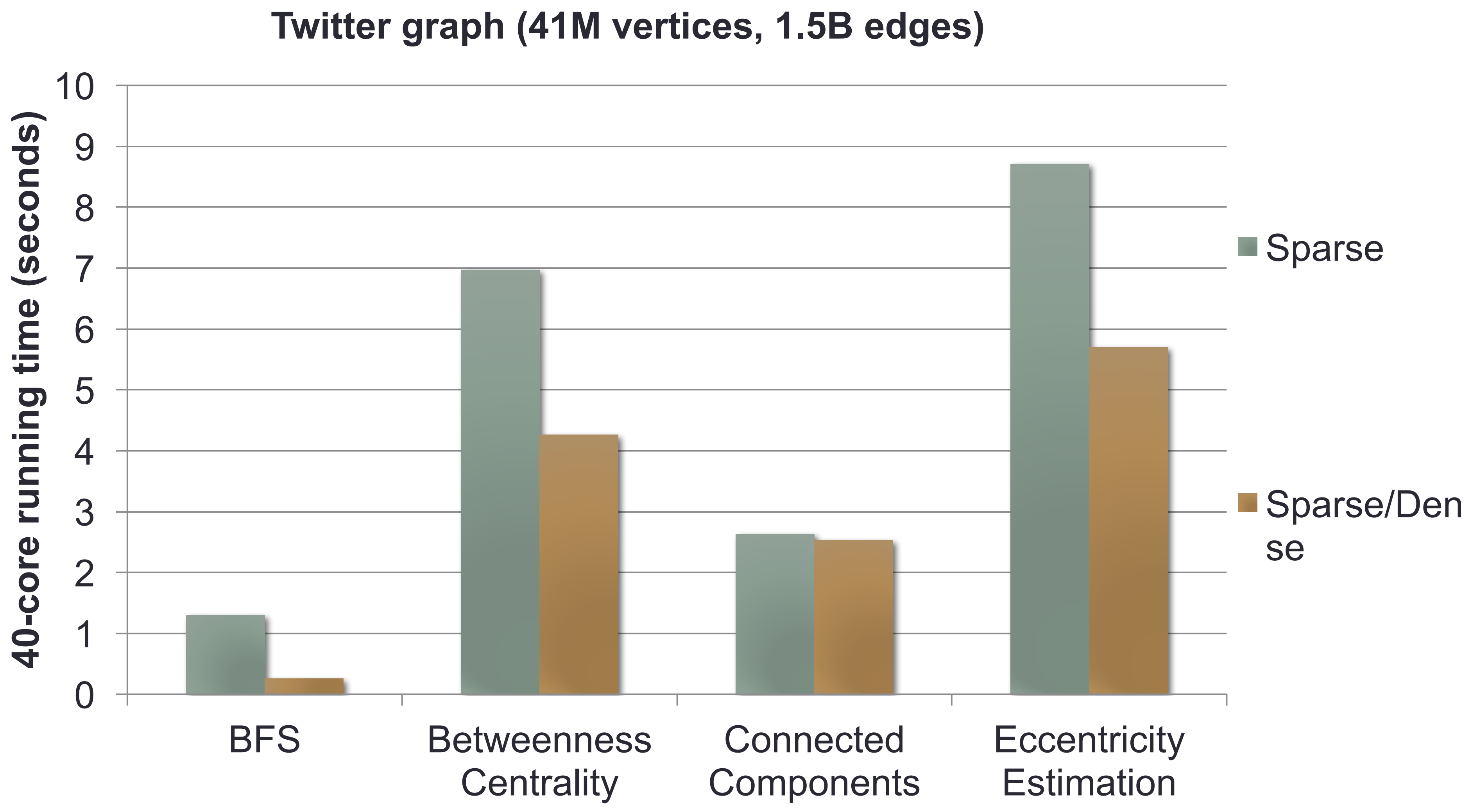


(e) PageRank-Delta

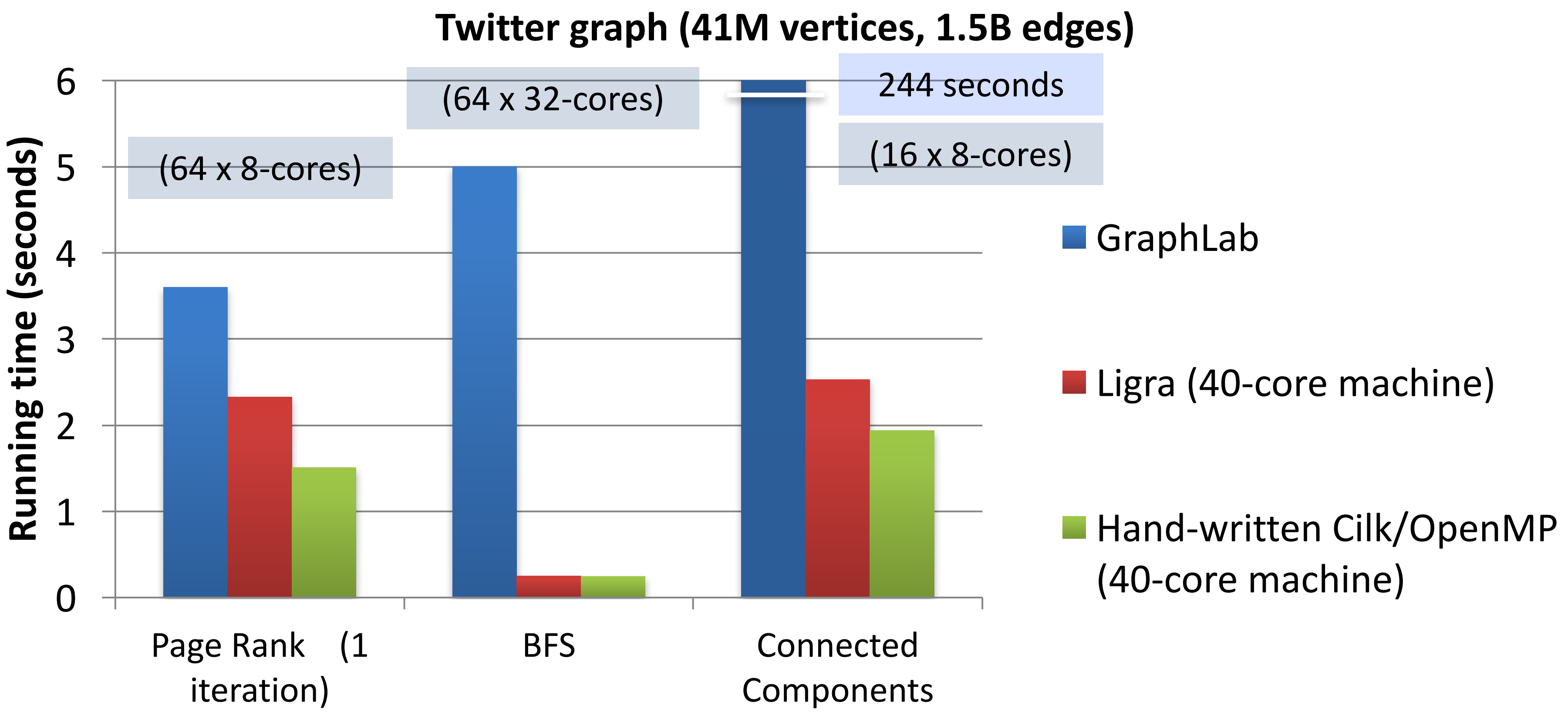


(f) Bellman-Ford

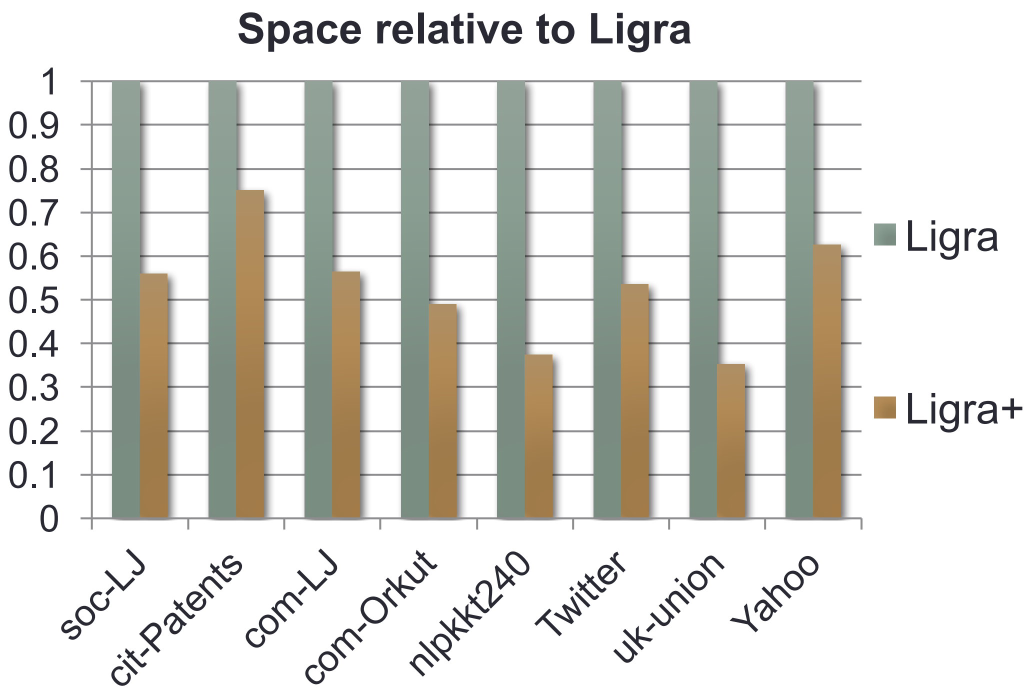
Benefit of Sparse/Dense Traversal

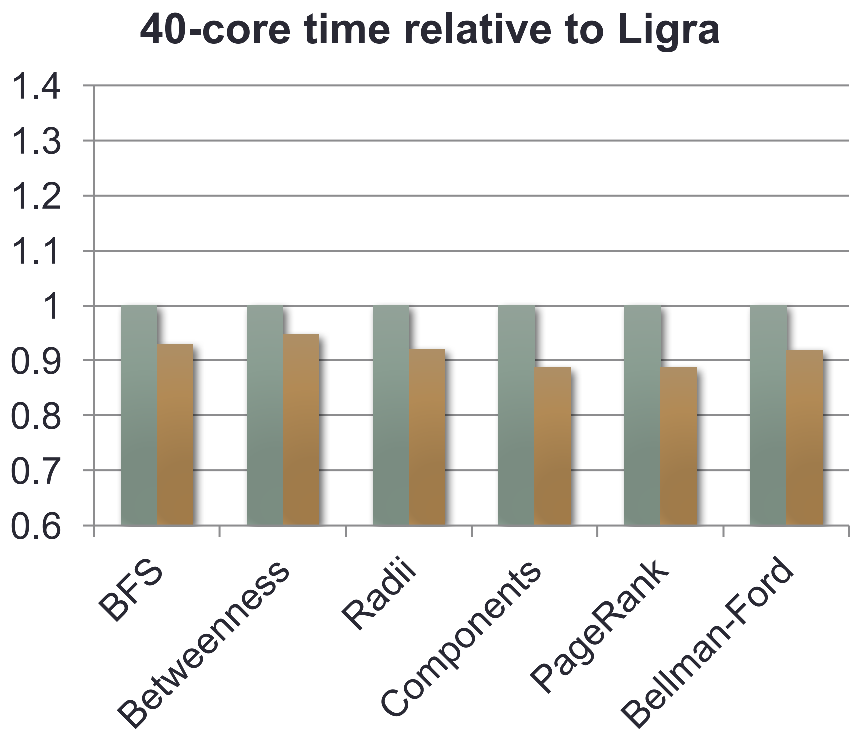


Ligra Performance



- Ligra performance close to **hand-written** code
- Faster than distributed-memory on per-core basis
- Several shared-memory graph processing systems subsequently developed: Galois [SOSP '13], X-stream [SOSP '13], PRISM [SPAA '14], Polymer [PPoPP '15], Ringo [SIGMOD '15]





- Cost of decoding on-the-fly?
- Memory bottleneck a bigger issue as graph algorithms are memory-bound