Some Sequential Algorithms are Almost Always Parallel

Guy Blelloch

Carnegie Mellon University

July 2017

Some Sequential Algorithms

?

are Almost Always

?

Parallel

?

Some Sequential Algorithms

?

are Almost Always

?

Parallel

?

Joint work with Jeremy Fineman, Phil Gibbons, Yan Gu, Julian Shun, and Yihan Sun [BFS SPAA'12], [BFGS PPOPP'12], [SGuBFG SODA'15], [BGuSSu SPAA'16].

Iterative Sequential Algorithms

```
for i = 1 to n
  do something
```

Iterative Sequential Algorithms

```
for i = 1 to n
  a[i] = f(a[i-1])
```

Iterative Sequential Algorithms

```
for i = 1 to n
    a[i] = f(a[i-1])
```

Fully Sequential (for arbitrary f)
Each iteration depends on all previous iterations.

Iterative Sequential Algorithms

```
for i = 1 to n
    a[i] = a[i] + 1
```

Iterative Sequential Algorithms

```
for i = 1 to n
    a[i] = a[i] + 1
```

Fully "parallel"

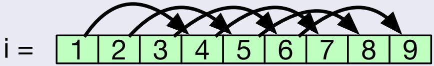
No dependences among iterations.

Iterative Sequential Algorithms

```
for i = $\sqrt{n}+1$ to $n$
    a[i] = f(a[i-$\sqrt{n}$])
```

Iterative Sequential Algorithms

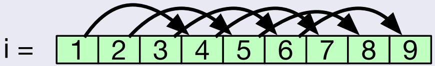
```
for i = $\sqrt{n}+1$ to $n$
    a[i] = f(a[i-$\sqrt{n}$])
```



$i \rightarrow j$ means j depends on i

Iterative Sequential Algorithms

```
for i = $\sqrt{n}+1$ to $n$
    a[i] = f(a[i-$\sqrt{n}$])
```



$i \rightarrow j$ means j depends on i

Partially parallel.

Some dependences, but they are not affected by the data.
In this case dependence depth is $\sqrt{n}$.

Iterative Sequential Algorithms

?

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
    H[$i$] = rand($\{0, \ldots, i\}$)
    swap(A[H[$i$]],A[$i$])
```

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

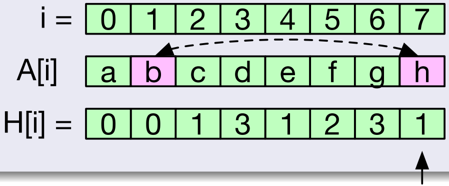
```
$A =$ an input array of length $n$
for $i = n-1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

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

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n-1$ downto 0
  H[$i$] = rand($\{0,\ldots,i\}$)
  swap(A[H[$i$]],A[$i$])
```

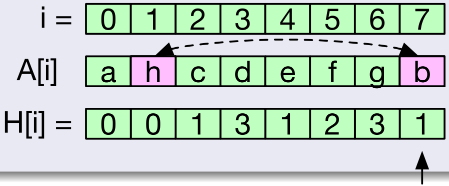


i =	0	1	2	3	4	5	6	7
A[i]	a	b	c	d	e	f	g	h
H[i] =	0	0	1	3	1	2	3	1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto $0$
  H[$i$] = rand($\{0, \dots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

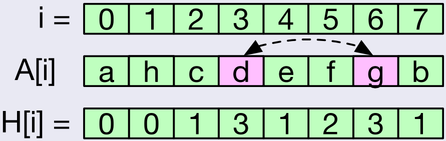


i =	0	1	2	3	4	5	6	7
A[i]	a	h	c	d	e	f	g	b
H[i] =	0	0	1	3	1	2	3	1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n-1$ downto 0
    H[$i$] = rand($\{0,\ldots,i\}$)
    swap(A[H[$i$]],A[$i$])
```

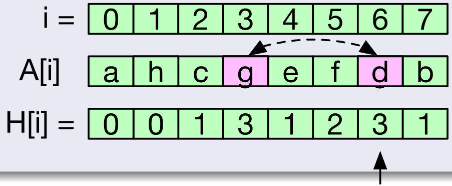


i =	0	1	2	3	4	5	6	7
A[i]	a	h	c	d	e	f	g	b
H[i] =	0	0	1	3	1	2	3	1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
**for** $i = n-1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```



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

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] $=$ rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

Dependences?

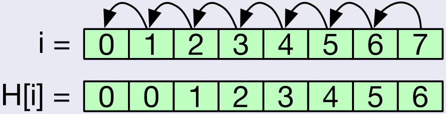
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	2	3	4	5	6

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

Fully sequential



i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	2	3	4	5	6

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
    H[$i$] = rand($\{0, \ldots, i\}$)
    swap(A[H[$i$]],A[$i$])
```

Dependences?

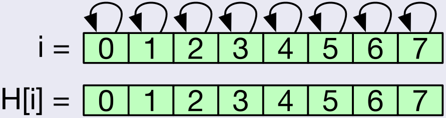
i =	0	1	2	3	4	5	6	7
H[i] =	0	1	2	3	4	5	6	7

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n-1$ downto 0
  H[$i$] $=$ rand($\{0,\ldots,i\}$)
  swap(A[H[$i$]],A[$i$])
```

Fully parallel



i = 0 1 2 3 4 5 6 7

H[i] = 0 1 2 3 4 5 6 7

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

In general?

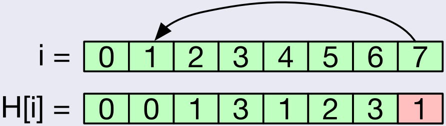
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] $=$ rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

In general?



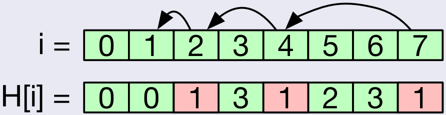
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] = rand($\{0, \ldots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

In general?



i = 0 1 2 3 4 5 6 7

H[i] = 0 0 1 3 1 2 3 1

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n-1$ downto 0
  H[$i$] $=$ rand($\{0,\ldots,i\}$)
  swap(A[H[$i$]],A[$i$])
```

Dependences **depend on data**.

Question: What can we say about dependence depth over the **random choices**?

Iterative Sequential Algorithms

Knuth's shuffle to generate a random permutation of A

```
$A =$ an input array of length $n$
for $i = n - 1$ downto 0
  H[$i$] $=$ rand($\{0, \dots, i\}$)
  swap(A[H[$i$]],A[$i$])
```

Dependences **depend on data**.

Question: What can we say about dependence depth over the **random choices**?

Answer [SGuBFG'15]: $O(\log n)$ w.h.p.

Iterative Sequential Algorithms

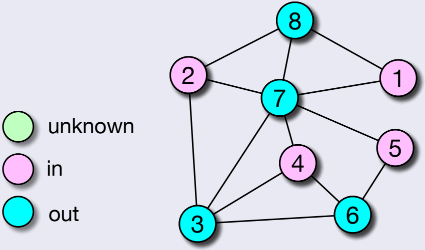
?

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \dots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

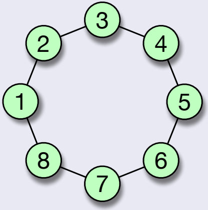


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n] =$ unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    then $S[i] =$ out else $S[i] =$ in
```

Dependences for this simple cycle graph?

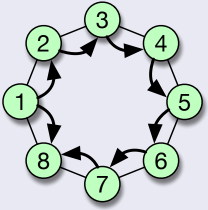


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Fully sequential

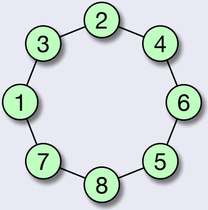


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n] =$ unknown
**for** $i = 1$ **to** $|V|$
    **if** for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    **then** $S[i] =$ out **else** $S[i] =$ in
```

Same graph, different ordering of V. Dependences now?

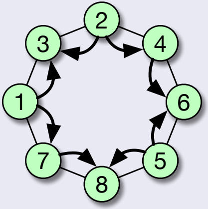


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$,  $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$,  $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Parallel?

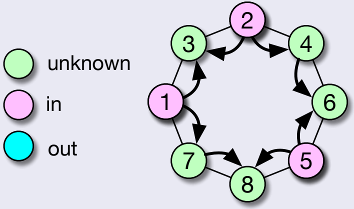


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Parallel?

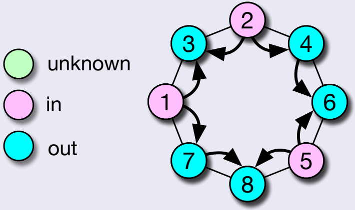


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n] =$ unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    then $S[i] =$ out else $S[i] =$ in
```

Fully parallel order (two rounds)

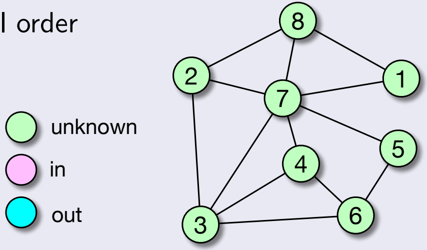


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \dots, n] =$ unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    then $S[i] =$ out else $S[i] =$ in
```

Partially parallel order

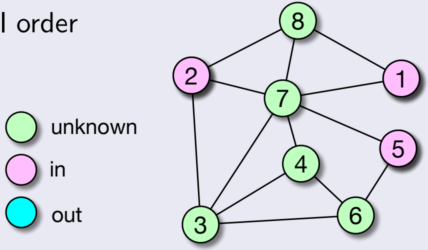


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Partially parallel order

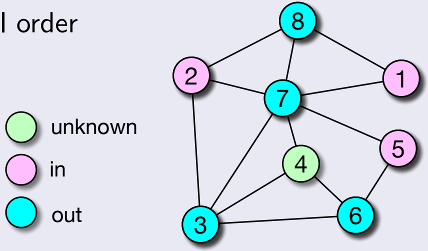


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Partially parallel order

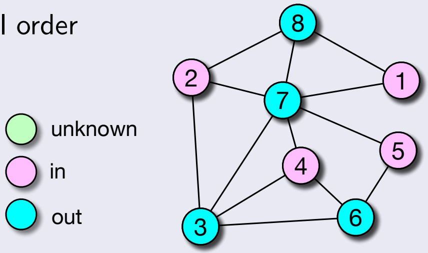


Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n] =$ unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    then $S[i] =$ out else $S[i] =$ in
```

Partially parallel order



Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Question: what is the dependence depth for a **random order of the vertices**?

Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Question: what is the dependence depth for a **random order of the vertices**?

Answer [BFS'12]: $O(\log^2 n)$
(w.h.p. for random ordering of V, i.e. if we randomly permute V)

Iterative Sequential Algorithms

Greedy Maximal Independent Set

```
undirected graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown
for $i = 1$ to $|V|$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Question: what is the dependence depth for a **random order of the vertices**?

Answer [BFS'12]: $O(\log^2 n)$
(w.h.p. for random ordering of V, i.e. if we randomly permute V)
Open problem: $O(\log n)$ w.h.p.?

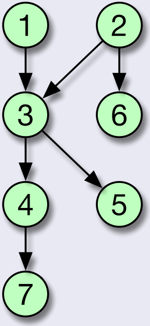
Dependence Depth

```
for i = 1 to n
    do something
```

Simple model:

- Each iterate is a vertex
- $i \to j$ means j depends on i

Iterations



Dependence Depth

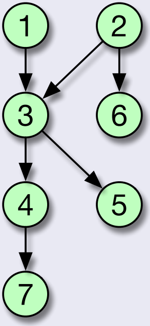
```
for i = 1 to n
    do something
```

Simple model:

- Each iterate is a vertex
- $i \rightarrow j$ means j depends on i

Iteration Depth:
the longest chain of dependences.

Iterations



Dependence Depth

```
for i = 1 to n
  do something
```

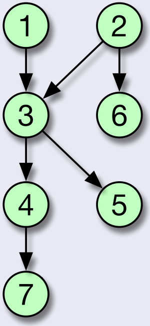
Simple model:

- Each iterate is a vertex
- $i \rightarrow j$ means j depends on i

Iteration Depth:
the longest chain of dependences.

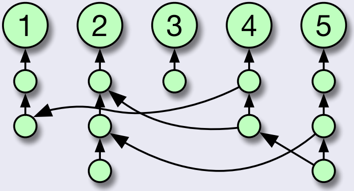
Overall Depth:
weighted by depth of each iteration.

Iterations



Dependence Depth

Nested iterations

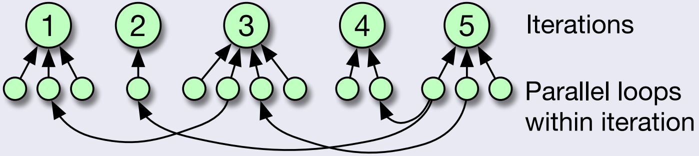


Iterations

Sequential loops
within iterations

Dependence Depth

Nested iterations



Only Two Examples?

1. Knuth shuffle
2. Greedy MIS

More Iterative Sequential Algorithms

Greedy Maximal Matching

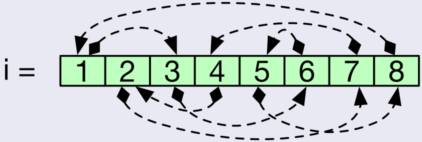
```
Graph $G = (V, E)$ and
$A[1, \ldots, |V|]$ = true        // $A[i]$ if vertex $i$ is available
$M[1, \ldots, |E|]$ = false       // $M[j]$ if edge $j$ is in matching
for $i = 1$ to $|E|$
    $(u, v) = E[i]$
    if ($A[u]$ and $A[v]$)
    then $M[i]$ = true, $A[u]$ = false, $A[v]$ = false
```

Iteration Depth [BFS'12]: $O(\log^2 |V|)$
(w.h.p. for random ordering of E)

More Iterative Sequential Algorithms

List contraction

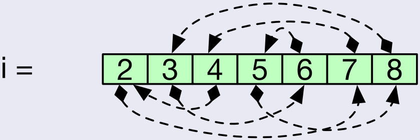
```
$A$ = an array containing $n$ links of doubly linked lists
for $i = 0$ to $n - 1$
    $A[i]$.next.previous = $A[i]$.previous
    $A[i]$.previous.next = $A[i]$.next
```



More Iterative Sequential Algorithms

List contraction

```
$A$ = an array containing $n$ links of doubly linked lists
for $i = 0$ to $n - 1$
    $A[i]$.next.previous = $A[i]$.previous
    $A[i]$.previous.next = $A[i]$.next
```



More Iterative Sequential Algorithms

List contraction

```
$A$ = an array containing $n$ links of doubly linked lists
for $i = 0$ to $n - 1$
    $A[i]$.next.previous = $A[i]$.previous
    $A[i]$.previous.next = $A[i]$.next
```

Applications of:

- length of lists or list ranking
- size or number of cycles in a permutation
- Euler tour, biconnectivity,..

More Iterative Sequential Algorithms

List contraction

```
$A$ = an array containing $n$ links of doubly linked lists
for $i = 0$ to $n - 1$
    $A[i]$.next.previous = $A[i]$.previous
    $A[i]$.previous.next = $A[i]$.next
```

Iteration Depth [SGuBFG'15]: $O(\log n)$
(w.h.p. for random ordering of A)

More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

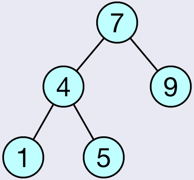
```
$A =$ an array of keys
$T$ = empty binary tree
for $i = 1$ to $n$
  BST_insert($T, A[i]$)
```

More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

```
$A$ = an array of keys
$T$ = empty binary tree
for $i = 1$ to $n$
    BST_insert($T, A[i]$)
```

Example:



insert(2), insert(8) – no dependence

insert(2), insert(3) – dependence

More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

```
$A$ = an array of keys
$T$ = empty binary tree
for $i = 1$ to $n$
  BST_insert($T, A[i]$)
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of A)

More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

```
$A =$ an array of keys
$T$ = empty binary tree
for $i = 1$ to $n$
  BST_insert($T, A[i]$)
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of A)

Overall Depth: $O(\log n)$

More Iterative Sequential Algorithms

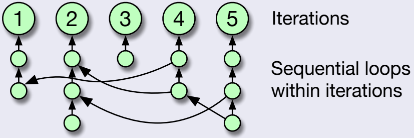
Sorting by insertion into a binary search tree (BST)

```
$A$ = an array of keys
$T$ = empty binary tree
for $i = 1$ to $n$
    BST_insert($T, A[i]$)
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of A)

Overall Depth: $O(\log n)$

Analysis requires sub-iteration dependences, otherwise $O(\log^2 n)$.



More Iterative Sequential Algorithms

two-dimensional linear programming

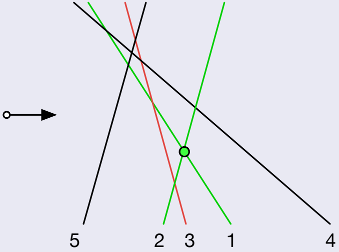
```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Example:

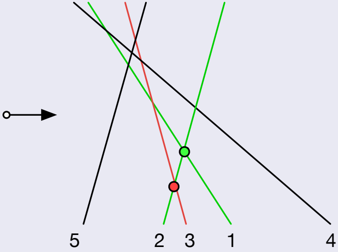


More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Example:

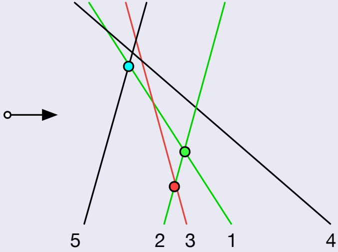


More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Example:



More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Total work: $O(n)$ (w.h.p. for random ordering of C)

More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Total work: $O(n)$ (w.h.p. for random ordering of C)

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of C)

More Iterative Sequential Algorithms

two-dimensional linear programming

```
Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
  if $p$ violates $c_i$
    $p$ = min intersection of $c_1, \ldots, c_{i-1}$ along $c_i$
```

Total work: $O(n)$ (w.h.p. for random ordering of C)

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of C)

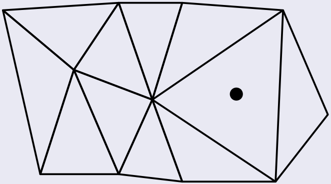
Overall Depth: $O(\log(n) \log\log(n))$
($\log\log(n)$ term for min intersection)

More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Example:

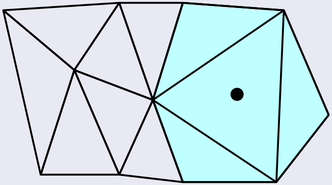


More Iterative Sequential Algorithms

Delaunay triangulation

```
points  $P = p_1, \ldots, p_n$
$T$  = {boundingTriangle of  $P$}
for  $i = 1$  to  $n$
  for  $t$  in  conflictSet($p_i$, $T$)
    replace  $t$  with new triangle(s) in  $T$
```

Example:

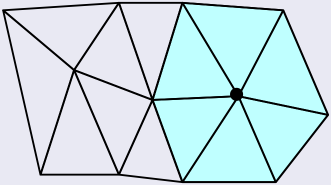


More Iterative Sequential Algorithms

Delaunay triangulation

```
points  $P = p_1, \ldots, p_n$
$T$  = {boundingTriangle of  $P$}
for  $i = 1$  to  $n$
  for  $t$  in  conflictSet($p_i$, $T$)
    replace  $t$  with new triangle(s) in  $T$
```

Example:

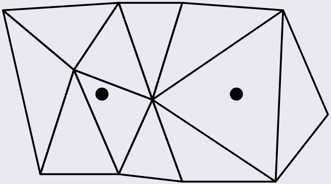


More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Example:

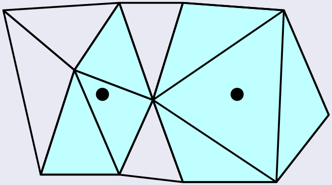


More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Example:

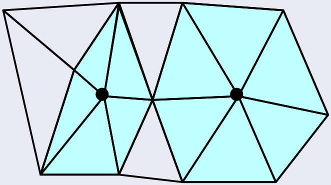


More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Example:



More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Total Work: $O(n \log n)$ (w.h.p. for random ordering of P)

More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of P)

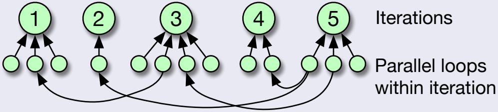
More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of P)

Analysis requires allowing subiterations to proceed independently.



More Iterative Sequential Algorithms

Delaunay triangulation

```
points $P = p_1, \ldots, p_n$
$T$ = {boundingTriangle of $P$}
for $i = 1$ to $n$
  for $t$ in conflictSet($p_i$, $T$)
    replace $t$ with new triangle(s) in $T$
```

Iteration Depth [BGuSSu'16]: $O(\log n)$
(w.h.p. for random ordering of P)

Parallel incremental Delaunay is **widely used in practice**, but it was not previously known whether it is theoretically efficient.

More Iterative Sequential Algorithms

Graph Connectivity using union-find

```
graph  $G = (V, E)$
F  =  a union find data structure on  $V$
for  $i = 1$  to  $|E|$
    $u$ = F.find($E[i].u$)
    $v$ = F.find($E[i].v$)
    if  ($u \neq v$)  then  F.union($u, v$)
```

More Iterative Sequential Algorithms

Graph Connectivity using union-find

```
graph $G = (V, E)$
F = a union find data structure on $V$
for $i = 1$ to $|E|$
    $u$ = F.find($E[i].u$)
    $v$ = F.find($E[i].v$)
    if ($u \neq v$) then F.union($u, v$)
```

Iteration Depth: TDB (open problem)

Some Sequential Algorithms

are Almost Always

Parallel

~~Some~~ Many Sequential Algorithms

- Iterative sequential algorithms such as:
  Knuth shuffle, greedy MIS, greedy maximal matching, list contraction, tree contraction, linear programming, Delaunay triangulation

are Almost Always

Parallel

~~Some~~ Many Sequential Algorithms

- Iterative sequential algorithms such as:
  Knuth shuffle, greedy MIS, greedy maximal matching, list contraction, tree contraction, linear programming, Delaunay triangulation

are Almost Always

- For almost all input orders (i.e. whp over random order)
  Or for almost all random choices (Knuth shuffle)

Parallel

~~Some~~ Many Sequential Algorithms

- Iterative sequential algorithms such as:
  Knuth shuffle, greedy MIS, greedy maximal matching, list contraction, tree contraction, linear programming, Delaunay triangulation

are Almost Always

- For almost all input orders (i.e. whp over random order)
  Or for almost all random choices (Knuth shuffle)

Parallel

- Polylogarithmic dependence depth

Why care?

Why care?

Intellectual curiosity

- Lots of work on parallel algorithms, but perhaps sequential ones are already parallel

Why care?

Intellectual curiosity

- Lots of work on parallel algorithms, but perhaps sequential ones are already parallel

Application to parallel and distributed algorithms (Theory)

- Surprisingly simple solutions to basic problems
- Possible way to attack new problems
- Theoretical justification to current practice

Why care?

Intellectual curiosity

- Lots of work on parallel algorithms, but perhaps sequential ones are already parallel

Application to parallel and distributed algorithms (Theory)

- Surprisingly simple solutions to basic problems
- Possible way to attack new problems
- Theoretical justification to current practice

Application to parallel and distributed algorithms (Practice)

- Fast and simple code
- Generic techniques to parallelize code
- Determinacy

How to Analyze Dependence Depth

Knuth Shuffle: Iteration Depth

```
for $i = n - 1$ downto 0
    H[$i$] = rand($\{0, \ldots, i\}$)
    swap(A[H[$i$]],A[$i$])
```

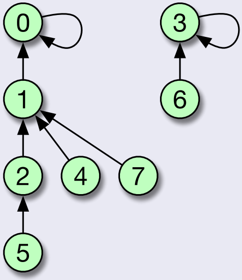
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1

Knuth Shuffle: Iteration Depth

i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1

Knuth Shuffle: Iteration Depth

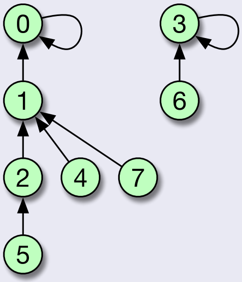
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1



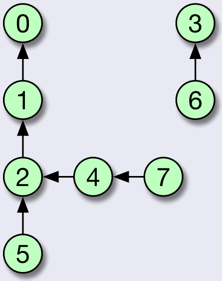
Chosen locations

Knuth Shuffle: Iteration Depth

i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1



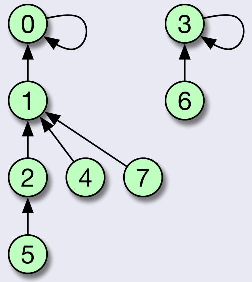
Chosen locations



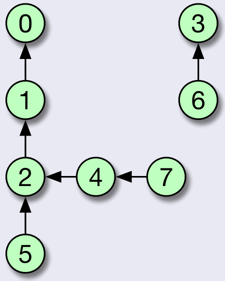
Actual dependences

Knuth Shuffle: Iteration Depth

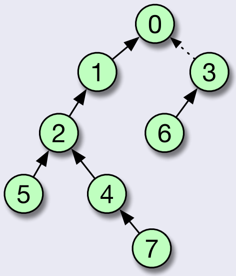
i =	0	1	2	3	4	5	6	7
H[i] =	0	0	1	3	1	2	3	1



Chosen locations



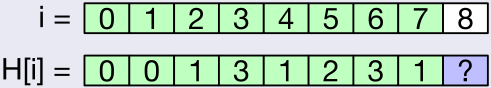
Actual dependences



Rearranged

Knuth Shuffle: Iteration Depth

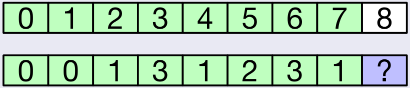
Adding one more, i.e., proof by induction



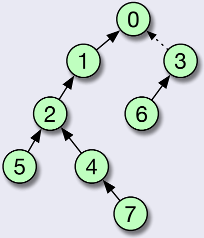
i =	0	1	2	3	4	5	6	7	8
H[i] =	0	0	1	3	1	2	3	1	?

Knuth Shuffle: Iteration Depth

Adding one more, i.e., proof by induction



i =	0	1	2	3	4	5	6	7	8
H[i] =	0	0	1	3	1	2	3	1	?

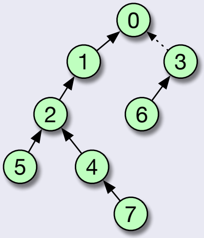


Not including 8

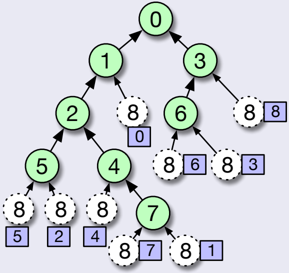
Knuth Shuffle: Iteration Depth

Adding one more, i.e., proof by induction

i =	0	1	2	3	4	5	6	7	8
H[i] =	0	0	1	3	1	2	3	1	?



Not including 8

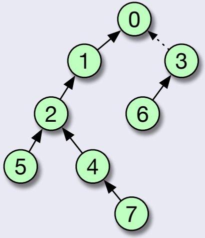


Possible positions of 8

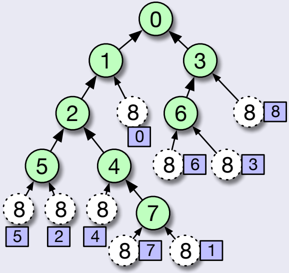
Knuth Shuffle: Iteration Depth

Adding one more, i.e., proof by induction

i =	0	1	2	3	4	5	6	7	8
H[i] =	0	0	1	3	1	2	3	1	?



Not including 8



Possible positions of 8

All equally likely, so equivalent to random BST.

Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown, $n = |V|$
for $i = 1$ to $n$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$,  $S[1, \ldots, n]$ = unknown,  $n = |V|$
for $i = 1$ to $n$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Known results

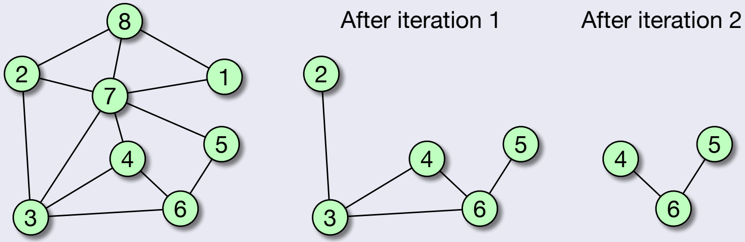
- Lexicographically first MIS is P-complete [Cook '85]
- $O(\log^2 n)$ dependence depth for random graphs w.h.p. [Coppersmith, Raghavan, Tompa '89]
- $O(\log n)$ depth for random graphs w.h.p. [Calkin, Frieze '90]
- $O(\log^2 n)$ depth for arbitrary graph in random order
- Many parallel algorithms (e.g. Luby).

Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$,  $S[1, \ldots, n] =$ unknown,  $n = |V|$
for $i = 1$ to $n$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j] =$ in
    then $S[i] =$ out else $S[i] =$ in
```

Definition (Residual graph on step *i*)

The graph that is left after step *i*



Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown, $n = |V|$
for $i = 1$ to $n$
  if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
  then $S[i]$ = out else $S[i]$ = in
```

Lemma (Degree)

After step i, the maximum degree in the residual graph is $O(n \log n / i)$ w.h.p. (over orderings of V).

Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$, $S[1, \ldots, n]$ = unknown, $n = |V|$
for $i = 1$ to $n$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Lemma (Degree)

After step i, the maximum degree in the residual graph is $O(n \log n / i)$ w.h.p. (over orderings of V).

For example:

For $i = n/2$ (half done), the max degree is $O(\log n)$ w.h.p.

Maximal Independent Set, Bound on Depth

```
graph $G = (V, E)$, $S[1, \dots, n]$ = unknown, $n = |V|$
for $i = 1$ to $n$
    if for any earlier neighbor $v_j$ of $v_i$, $S[j]$ = in
    then $S[i]$ = out else $S[i]$ = in
```

Lemma (Degree)

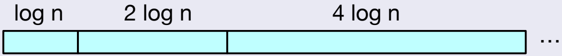
After step i, the maximum degree in the residual graph is $O(n \log n / i)$ w.h.p. (over orderings of V).

Proof outline.

Consider a vertex with degree larger than d (in residual graph) on step i. The probability of selecting one of the neighbors on each step j ($\leq i$) is at least d/n. The probability it survives all steps j is therefore at most $(1 - d/n)^i$, leading to the result. ☐

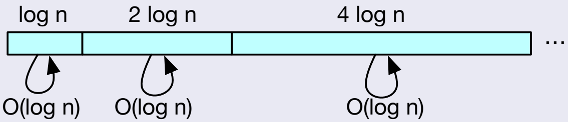
Maximal Independent Set, Bound on Depth

Consider increasing sized blocks of the iterations



Maximal Independent Set, Bound on Depth

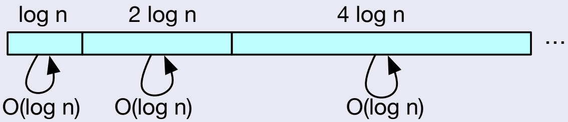
Consider increasing sized blocks of the iterations



w.h.p. no path within a block is greater than $O(\log n)$

Maximal Independent Set, Bound on Depth

Consider increasing sized blocks of the iterations



Proof outline: For each block *i*

- Prob. of edge beween two iterations is at most $\frac{1}{2^i}$ (by Degree Lemma)
- number of paths of length l is $\binom{2^i logn}{l}$

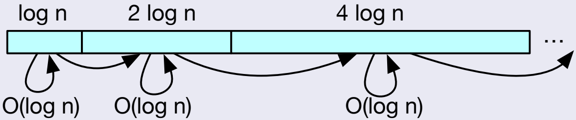
By the union bound the prob. of any path of length l is at most

$$\left(\frac{1}{2^i}\right)^l \binom{2^i logn}{l} < \left(\frac{1}{2^i}\right)^l \left(\frac{e2^i \log n}{l}\right)^l = \left(\frac{e \log n}{l}\right)^l$$

Therefore probability is very small that $l > 2e \log n$.

Maximal Independent Set, Bound on Depth

Consider increasing sized blocks of the iterations



Summary

Since path within each block is $O(\log n)$, and number of blocks is $O(\log n)$, total depth is $O(\log^2 n)$.

Can be improved to $O(\log n \log d_{max})$ by picking blocks of size $2^i \frac{d_{max}}{n} \log n$.

Concrete Algorithms and Implementation

Concrete Algorithms and Implementation

Definition (Efficiently Checkable Dependences)

In a constant number of “rounds” each iteration can check if it has any unresolved dependences.

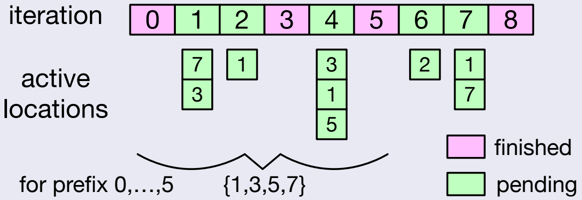
Concrete Algorithms and Implementation

Definition (Efficiently Checkable Dependences)

In a constant number of “rounds” each iteration can check if it has any unresolved dependences.

Sufficient condition (true for all our examples):

1. Each i independently can identify *active* locations l_i, s.t.,
2. pending iterations in every prefix $0, \ldots, i$ only update $\cup_{j \in [i]} l_j$.



Concrete Algorithms and Implementation

Definition (Efficiently Checkable Dependences)

In a constant number of “rounds” each iteration can check if it has any unresolved dependences.

Sufficient condition (true for all our examples):

1. Each i independently can identify *active* locations l_i, s.t.,
2. pending iterations in every prefix $0, \ldots, i$ only update $\cup_{j \in [i]} l_j$.

Suggests an implementation strategy:

While there are pending iterations:

reserve: in parallel pending iterations find active locations and mark them

commit: in parallel each pending iteration runs, but aborts if it depends on an active previous location

Implication For Algorithms

Parallel (Priority PRAM)

	Work	Depth (Time)
MIS	$O(\|E\|)$	$O(\log^2 \|V\| \log \Delta)$
Maximal Matching	$O(\|E\|)$	$O(\log^2 \|V\| \log \Delta)$
BST Sort	$O(n \log n)$	$O(\log n)$
Knuth Shuffle	$O(n)$	$O(\log n \log^* n)$
List Contraction	$O(n)$	$O(\log n \log^* n)$
2d Linear Programming	$O(n)$	$O(\log n)$
Delaunay triangulation	$O(n \log n)$	$O(\log^2 n)$

All are work efficient. Δ = maximum degree

Implication For Algorithms

Parallel (Priority PRAM)

	Work	Depth (Time)
MIS	$O(\|E\|)$	$O(\log^2 \|V\| \log \Delta)$
Maximal Matching	$O(\|E\|)$	$O(\log^2 \|V\| \log \Delta)$
BST Sort	$O(n \log n)$	$O(\log n)$
Knuth Shuffle	$O(n)$	$O(\log n \log^* n)$
List Contraction	$O(n)$	$O(\log n \log^* n)$
2d Linear Programming	$O(n)$	$O(\log n)$
Delaunay triangulation	$O(n \log n)$	$O(\log^2 n)$

All are work efficient. $\Delta =$ maximum degree

Distributed (CONGEST model)

	Rounds
MIS	$O(\log \|V\| \log \Delta)$
Maximal Matching	$O(\log \|V\| \log \Delta)$

Implementation: Speculative For

```
struct step {
  bool reserve(int i) {
    reserves locations that will be written by iteration $i$}
  bool commit(int i) {
    checks locations iteration $i$ depends on, and runs if safe}};
```

Implementation: Speculative For

```
struct step {
  bool reserve(int i) {
    reserves locations that will be written by iteration $i$}
  bool commit(int i) {
    checks locations iteration $i$ depends on, and runs if safe}};
```

```
speculative_for(Step(..), 0, n);  // $0, \ldots, n =$ range of iterations
```

Implementation: Speculative For

```
struct step {
  bool reserve(int i) {
    reserves locations that will be written by iteration i}
  bool commit(int i) {
    checks locations iteration i depends on, and runs if safe}};
```

```
speculative_for(Step(..), 0, n);  // 0, . . . , n = range of iterations
```

`speculative_for` will repeat the following until done:

- picks a prefix of remaining (pending) iterations
- in parallel runs the `reserve` on the prefix
- in parallel runs the `commit` on the prefix
- removes completed iterations (`commit` returns 1)

Implementation: Speculative For

```
struct step {
  bool reserve(int i) {
    reserves locations that will be written by iteration i}
  bool commit(int i) {
    checks locations iteration i depends on, and runs if safe}};
```

```
speculative_for(Step(..), 0, n);   // 0, . . . , n = range of iterations
```

`speculative_for` will repeat the following until done:

- picks a prefix of remaining (pending) iterations
- in parallel runs the `reserve` on the prefix
- in parallel runs the `commit` on the prefix
- removes completed iterations (`commit` returns 1)

Can dynamically choose size of prefix.

Knuth Shuffle Code

```
struct knuth_step {
  bool reserve(int i) {
    write_min(R[i], i); write_min(R[H[i]], i);
    return 1; }
  bool commit (int i) {
    int h = H[i];
    if(R[H[i]] == i) {
      if(R[i] == i) {swap(A[i],A[H[i]]); R[i] = inf; return 1;}
      R[H[i]] = inf;}
    return 0; }
};
```

```
speculative_for(knuth_step(..), 0, n);
```

MIS Code

```
struct mis_step {
  bool reserve(int i) {
    flag = In;
    for (int j = 0; j < G[i].degree; j++) {
      int ngh = G[i].Neighbors[j];
      if (ngh < i) {
        if (S[ngh] == In) { flag = Out; return 1;}
        else if (S[ngh] == Unknown) flag = Unknown; }}
    return 1;}

  bool commit(int i) { return (S[i] = flag) != Unknown;}
};
```

```
speculative_for(mis_step(..), 0, n);
```

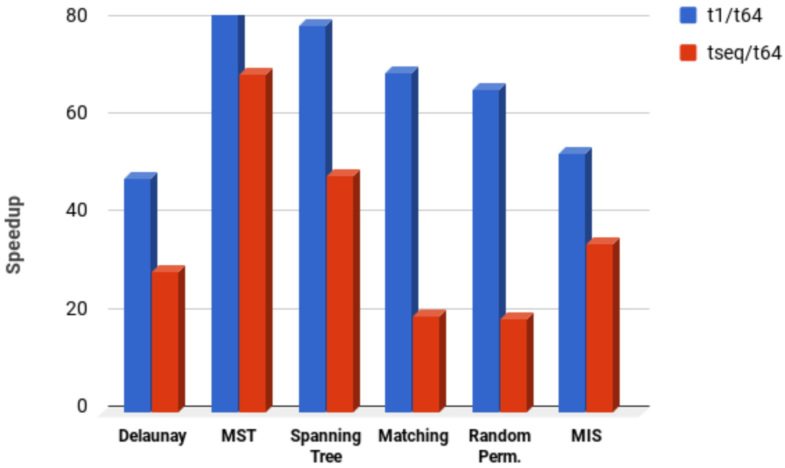
Spanning Tree Code

```
struct union_find_step {
  bool reserve(int i) {
    u = UF.find(E[i].u);
    v = UF.find(E[i].v);
    if (u > v) swap(u,v);
    if (u != v) { write_min(R[v], i); return 1;
    } else return 0; }

  bool commit(int i) {
    if (R[v] == i) { UF.link(v, u); return 1; }
    else return 0; }
};
```

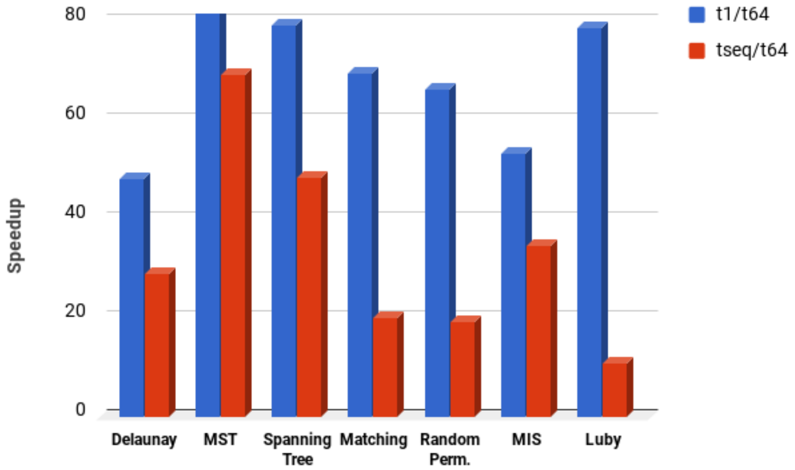
```
speculative_for(union_find_step(..), 0, m);
```

Timings on a 64-core Xeon Phi



tseq = best sequential algorithm
t1 = time on one core
t64 = time on all cores

Timings on a 64-core Xeon Phi



tseq = best sequential algorithm
t1 = time on one core
t64 = time on all cores

End Matter

Conclusions

1. Many sequential algorithms are “inherently” parallel, at least when randomly ordering.
   - Perhaps should be thinking of algorithms more abstractly in terms of their dependence graph instead of specific model.
2. Can often take advantage of the parallelism using reservations
3. Resulting code is **simple**, **fast**, and **deterministic**

End Matter

Conclusions

1. Many sequential algorithms are “inherently” parallel, at least when randomly ordering.
    - Perhaps should be thinking of algorithms more abstractly in terms of their dependence graph instead of specific model.
2. Can often take advantage of the parallelism using reservations
3. Resulting code is **simple**, **fast**, and **deterministic**

Open Questions

1. Depth of MIS
2. Resolving dependences in a more general context.