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].
for $i = 1$ to $n$
    do something
Iterative Sequential Algorithms

for i = 1 to n
    a[i] = f(a[i-1])
Iterative Sequential Algorithms

\[
\text{for } i = 1 \text{ to } n \\
\quad a[i] = f(a[i-1])
\]

**Fully Sequential** (for arbitrary \( f \))
Each iteration depends on all previous iterations.
for $i = 1$ to $n$
\quad $a[i] = a[i] + 1$
Iterative Sequential Algorithms

for \( i = 1 \) to \( n \)
\[
a[i] = a[i] + 1
\]

Fully “parallel”
No dependences among iterations.
for $i = \sqrt{n} + 1$ to $n$

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

```
i = 1 2 3 4 5 6 7 8 9
```

*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}*. 

SPAA/PODC 2017
Some Sequential Algorithms are Almost Always Parallel 6 / 35
\[ A = \text{an input array of length } n \]

\begin{align*}
\textbf{for } i &= n - 1 \text{ downto } 0 \\
H[i] &= \text{rand}\{0, \ldots, i\} \\
\text{swap}(A[H[i]], A[i])
\end{align*}
Knuth’s shuffle to generate a random permutation of $A$

\[ A = \text{an input array of length } n \]
\[ \text{for } i = n - 1 \text{ downto } 0 \]
\[ H[i] = \text{rand}(\{0, \ldots, i\}) \]
\[ \text{swap}(A[H[i]],A[i]) \]
Knuth’s shuffle to generate a random permutation of $A$

$A$ = an input array of length $n$

for $i = n - 1$ downto 0

$H[i] = \text{rand}([0, \ldots, i])$

swap($A[H[i]], A[i]$)

---

$i = 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7$

$A[i] = a \ b \ c \ d \ e \ f \ g \ h$

$H[i] = 0 \ 0 \ 1 \ 3 \ 1 \ 2 \ 3 \ 1$
Knuth’s shuffle to generate a random permutation of \( A \)

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

\[
\begin{array}{l}
i = 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \\
\text{A[i]} = \begin{array}{cccccccc}
\text{a} & \text{b} & \text{c} & \text{d} & \text{e} & \text{f} & \text{g} & \text{h}
\end{array} \\
\text{H[i]} = 0 \ 0 \ 1 \ 3 \ 1 \ 2 \ 3 \ 1
\end{array}
\]
Knuth’s shuffle to generate a random permutation of $A$

$$A = \text{an input array of length } n$$

for $i = n - 1$ downto 0

$$H[i] = \text{rand}\{0, \ldots, i\}$$

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 $A$

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

\[ i = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \]
\[ A[i] = \begin{bmatrix} a & h & c & d & e & f & g & b \end{bmatrix} \]
\[ H[i] = \begin{bmatrix} 0 & 0 & 1 & 3 & 1 & 2 & 3 & 1 \end{bmatrix} \]
**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
    $H[i] = \text{rand}([0, \ldots, i])$
    swap($A[H[i]], A[i]$)
```

---

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

\[ A[i] = a \quad h \quad c \quad g \quad e \quad f \quad d \quad b \]

\[ H[i] = 0 \quad 0 \quad 1 \quad 3 \quad 1 \quad 2 \quad 3 \quad 1 \]
Knuth’s shuffle to generate a random permutation of $A$

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

Dependences?

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

\[
\begin{array}{cccccccc}
  i &=& 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  H[i] &=& 0 & 0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]
Knuth’s shuffle to generate a random permutation of $A$

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

Dependences?

\[
i = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \\
H[i] = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}
\]
Knuth’s shuffle to generate a random permutation of $A$

$A = \text{an input array of length } n$

\text{for } i = n - 1 \text{ downto } 0$

\begin{align*}
\text{H}[i] & = \text{rand}(\{0, \ldots, i\}) \\
\text{swap}(A[\text{H}[i]], A[i])
\end{align*}

Fully parallel

\begin{align*}
\text{i} & = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix} \\
\text{H}[i] & = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}
\end{align*}
Knuth’s shuffle to generate a random permutation of $A$

$A = \text{an input array of length } n$

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

In general?

\begin{verbatim}
i = 0 1 2 3 4 5 6 7
H[i] = 0 0 1 3 1 2 3 1
\end{verbatim}
Knuth’s shuffle to generate a random permutation of $A$

$A =$ an input array of length $n$

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

In general?

\[
\begin{array}{c}
\text{i} = 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\text{H}[i] = 0 & 0 & 1 & 3 & 1 & 2 & 3 & 1
\end{array}
\]
Knuth’s shuffle to generate a random permutation of $A$

$A =$ an input array of length $n$

for $i = n - 1$ downto 0

$H[i] = \text{rand}([0, \ldots, i])$

swap($A[H[i]], A[i]$)

In general?

\[
\begin{array}{l}
i = 0 \, 1 \, 2 \, 3 \, 4 \, 5 \, 6 \, 7 \\
H[i] = 0 \, 0 \, 1 \, 3 \, 1 \, 2 \, 3 \, 1 
\end{array}
\]
Knuth’s shuffle to generate a random permutation of $A$

$$A = \text{an input array of length } n$$

**for** $i = n - 1 \text{ downto } 0$

$$H[i] = \text{rand}([0, \ldots, i})$$

$$\text{swap}(A[H[i]], A[i])$$

Dependences depend on data.

**Question:** What can we say about dependence depth over the random choices?
Knuth’s shuffle to generate a random permutation of $A$

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

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

    then $S[i] = \text{out}$

else $S[i] = \text{in}$
Greedy Maximal Independent Set

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

Dependences for this simple cycle graph?
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|$

if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$

then $S[i] = \text{out}$ else $S[i] = \text{in}$
**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 \( v_i \), \( S[j] = \text{in} \)

then \( S[i] = \text{out} \) else \( S[i] = \text{in} \)

Same graph, different ordering of \( V \). Dependences now?
Greedy Maximal Independent Set

undirected graph \( G = (V, E) \), \( S[1, \ldots, n] = \text{unknown} \)

for \( i = 1 \) to \( |V| \)
    if for any earlier neighbor \( v_j \) of \( v_i \), \( S[j] = \text{in} \)
    then \( S[i] = \text{out} \) else \( S[i] = \text{in} \)
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|
  \quad$ if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$
  \quad then $S[i] = \text{out}$ else $S[i] = \text{in}$
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)

![Diagram of a graph with nodes numbered 1 to 8, showing the order of processing and their states as unknown, in, or out.](image)
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 $v_i$, $S[j] = \text{in}$
  then $S[i] = \text{out}$ else $S[i] = \text{in}$

Partially parallel order

![Graph Diagram]

- unknown
- in
- out
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|$  
  if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$  
  then $S[i] = \text{out}$  
  else $S[i] = \text{in}$

Partially parallel order
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|$
  if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$
  then $S[i] = \text{out}$ else $S[i] = \text{in}$

Partially parallel order
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|
    \text{if for any earlier neighbor } v_j \text{ of } v_i, \ S[j] = \text{in}
    \text{then } S[i] = \text{out} \ \text{else } S[i] = \text{in}$

Partially parallel order

```
unknown
in
out
```

```
8 7 6 5 4 3 2 1
```

---

Iterative Sequential Algorithms

Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|
    \text{if for any earlier neighbor } v_j \text{ of } v_i, \ S[j] = \text{in}
    \text{then } S[i] = \text{out} \ \text{else } S[i] = \text{in}$

Partially parallel order

```
unknown
in
out
```

```
8 7 6 5 4 3 2 1
```
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?
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 \( v_i \), \( S[j] = \text{in} \)
  then \( S[i] = \text{out} \) else \( S[i] = \text{in} \)

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 \( V \))
Greedy Maximal Independent Set

undirected graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$

for $i = 1$ to $|V|$
  if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$
  then $S[i] = \text{out}$
  else $S[i] = \text{in}$

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.?
for i = 1 to n
do something

Simple model:
- Each iterate is a vertex
- \( i \rightarrow j \) means \( j \) depends on \( i \)
for \( i = 1 \) to \( n \)

\[ \text{do something} \]

**Simple model:**
- Each iterate is a vertex
- \( i \rightarrow j \) means \( j \) depends on \( i \)

**Iteration Depth:**
the longest chain of dependences.
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.
Dependence Depth

Nested iterations

Sequential loops within iterations

Iterations

1 2 3 4 5
Dependence Depth

Nested iterations

1 2 3 4 5

Parallel loops within iteration

Iterations
Only Two Examples?

1. Knuth shuffle
2. Greedy MIS
Greedy Maximal Matching

Graph $G = (V, E)$ and

$A[1, \ldots, |V|] = \text{true}$  \hspace{1em} // $A[i]$ if vertex $i$ is available

$M[1, \ldots, |E|] = \text{false}$  \hspace{1em} // $M[j]$ if edge $j$ is in matching

for $i = 1$ to $|E|$

$(u, v) = E[i]$


then $M[i] = \text{true}$, $A[u] = \text{false}$, $A[v] = \text{false}$

Iteration Depth [BFS’12]: $O(\log^2 |V|)$

(w.h.p. for random ordering of $E$)
List contraction

\[
A = \text{an array containing } n \text{ links of doubly linked lists}
\]

\[
\text{for } i = 0 \text{ to } n - 1 \\
\quad A[i].\text{next}.\text{previous} = A[i].\text{previous} \\\n\quad A[i].\text{previous}.\text{next} = A[i].\text{next}
\]
List contraction

\[ A = \text{an array containing } n \text{ links of doubly linked lists} \]

\[ \text{for } i = 0 \text{ to } n - 1 \]

\[ A[i].\text{next.previous} = A[i].\text{previous} \]

\[ A[i].\text{previous.next} = A[i].\text{next} \]
List contraction

\[ A = \text{an array containing } n \text{ links of doubly linked lists} \]
\[ \text{for } i = 0 \text{ to } n - 1 \]
\[ A[i].\text{next.previous} = A[i].\text{previous} \]
\[ A[i].\text{previous.next} = A[i].\text{next} \]

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

\[ A = \text{an array containing } n \text{ links of doubly linked lists} \]

\[ \text{for } i = 0 \text{ to } n - 1 \]
\[ A[i].\text{next.previous} = A[i].\text{previous} \]
\[ A[i].\text{previous.next} = A[i].\text{next} \]

Iteration Depth [SGuBFG’15]: \( O(\log n) \)
(w.h.p. for random ordering of \( A \))
Sorting by insertion into a binary search tree (BST)

\[
A = \text{an array of keys} \\
T = \text{empty binary tree} \\
\text{for } i = 1 \text{ to } n \\
\quad \text{BST_insert}(T, A[i])
\]
More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

\[ A = \text{an array of keys} \]
\[ T = \text{empty binary tree} \]
\[ \text{for } i = 1 \text{ to } n \]
\[ \text{BST}_{\text{insert}}(T, A[i]) \]

Example:

```
7
4 9
1 5
```

- insert(2), insert(8) – no dependence
- insert(2), insert(3) – dependence
More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

\[ A = \text{an array of keys} \]
\[ T = \text{empty binary tree} \]
\[ \textbf{for } i = 1 \text{ to } n \]
\[ \quad \text{BST\textunderscore insert}(T, A[i]) \]

\textbf{Iteration Depth} [BGuSSu’16]: \( O(\log n) \)
\( \text{w.h.p. for random ordering of } A \)
More Iterative Sequential Algorithms

Sorting by insertion into a binary search tree (BST)

\[ A = \text{an array of keys} \]
\[ T = \text{empty binary tree} \]
\[ \text{for } i = 1 \text{ to } n \]
\[ \text{BST_insert}(T, A[i]) \]

**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
\( \text{for } i = 1 \text{ to } n \)
\[ \text{BST}_\text{insert}(T, A[i]) \]

\textbf{Iteration Depth} [BGuSSu’16]: \( O(\log n) \)
(w.h.p. for random ordering of \( A \))

\textbf{Overall Depth}: \( O(\log n) \)
Analysis requires
sub-iteration dependences, otherwise
\( O(\log^2 n) \).
two-dimensional linear programming

Constraints (lines) $C = c_1, \ldots, c_n$
$p = (\infty, 0)$
for $i = 1$ to $n$
    if $p$ violates $c_i$
    then $p = \text{min intersection of } c_1, \ldots, c_{i-1} \text{ along } c_i$
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:
two-dimensional linear programming

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

Example:
two-dimensional linear programming

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

Example:
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 \( C \))
two-dimensional linear programming

Constraints (lines) $C = c_1, \ldots, c_n$

$p = (\infty, 0)$

for $i = 1$ to $n$

if $p$ violates $c_i$

$p = \text{min intersection of } c_1, \ldots, c_{i-1} \text{ along } c_i$

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)
Delaunay triangulation

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

Example:
Delaunay triangulation

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

Example:
Delaunay triangulation

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

Example:
Delaunay triangulation

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

Example:
Delaunay triangulation

points $P = p_1, \ldots, p_n$

$T = \{\text{boundingTriangle of } P\}$

for $i = 1 \text{ to } n$

for $t$ in conflictSet($p_i, T$)

replace $t$ with new triangle(s) in $T$

Example:
More Iterative Sequential Algorithms

Delaunay triangulation

points $P = p_1, \ldots, p_n$

$T = \{\text{boundingTriangle of } P\}$

for $i = 1$ to $n$

for $t$ in conflictSet($p_i, T$)

replace $t$ with new triangle(s) in $T$

Example:
Delaunay triangulation

points \( P = p_1, \ldots, p_n \)

\( T = \{\text{boundingTriangle of } P\} \)

for \( i = 1 \) to \( n \)

for \( t \) in \( \text{conflictSet}(p_i, T) \)

replace \( t \) with new triangle(s) in \( T \)

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

points $P = p_1, \ldots, p_n$
$T = \{\text{boundingTriangle of } P\}$
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 $P$)
Delaunay triangulation

points \( P = p_1, \ldots, p_n \)
\( T = \{\text{boundingTriangle of } P\} \)
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.

Iterations

Parallel loops within iteration
More Iterative Sequential Algorithms

Delaunay triangulation

points $P = p_1, \ldots, p_n$
$T = \{\text{boundingTriangle of } P\}$
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 \textbf{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 $F.union(u, v)$
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?

- 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
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
for $i = n - 1$ downto 0

$H[i] = \text{rand}([0,\ldots,i])$

swap(A[H[i]], A[i])
Knuth Shuffle: Iteration Depth

\[ i = \begin{array}{l}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ H[i] = \begin{array}{l}
0 & 0 & 1 & 3 & 1 & 2 & 3 & 1 \\
\end{array} \]
### Knuth Shuffle: Iteration Depth

- **i:** 
  \[
  i = \begin{array}{cccccccc}
  0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
  \end{array}
  \]

- **H[i]:** 
  \[
  H[i] = \begin{array}{cccccccc}
  0 & 0 & 1 & 3 & 1 & 2 & 3 & 1 \\
  \end{array}
  \]

#### Chosen locations
- 0
- 3
- 1
- 7
- 2
- 4
- 5
- 6

#### Actual dependences

#### Rearranged
Knuth Shuffle: Iteration Depth

\[
i = \begin{bmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{bmatrix}
\]

\[
H[i] = \begin{bmatrix} 0 & 0 & 1 & 3 & 1 & 2 & 3 & 1 \end{bmatrix}
\]

Chosen locations

Actual dependences
Knuth Shuffle: Iteration Depth

\[
i = \begin{bmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{bmatrix}
\]

\[
H[i] = \begin{bmatrix}
0 & 0 & 1 & 3 & 1 & 2 & 3 & 1
\end{bmatrix}
\]

Chosen locations

Actual dependences

Rearranged
Knuth Shuffle: Iteration Depth

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

\[ i = \boxed{0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8} \]
\[ H[i] = \boxed{0 \ 0 \ 1 \ 3 \ 1 \ 2 \ 3 \ 1 \ ?} \]
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.
Knuth Shuffle: Iteration Depth

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

\[
i = 0 \quad 1 \quad 2 \quad 3 \quad 4 \quad 5 \quad 6 \quad 7 \quad 8
\]

\[
H[i] = 0 \quad 0 \quad 1 \quad 3 \quad 1 \quad 2 \quad 3 \quad 1 \quad ?
\]

Not including 8

Possible positions of 8

Not including 8
Knuth Shuffle: Iteration Depth

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

\[
i = \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\end{array}
\]

\[
H[i] = \begin{array}{cccccccc}
0 & 0 & 1 & 3 & 1 & 2 & 3 & 1 & ? \\
\end{array}
\]

Not including 8

Possible positions of 8

All equally likely, so equivalent to random BST.
graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$, $n = |V|$ 

for $i = 1$ to $n$

if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$

then $S[i] = \text{out}$ else $S[i] = \text{in}$
Maximal Independent Set, Bound on Depth

Graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$, $n = |V|$

For $i = 1$ to $n$
  If for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$
  Then $S[i] = \text{out}$
  Else $S[i] = \text{in}$

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 \( G = (V, E) \), \( S[1, \ldots, n] = \text{unknown} \), \( n = |V| \)

\( \text{for } i = 1 \text{ to } n \)
\( \text{if } \) for any earlier neighbor \( v_j \) of \( v_i \), \( S[j] = \text{in} \)
\( \text{then } S[i] = \text{out} \text{ else } S[i] = \text{in} \)

**Definition (Residual graph on step } i \) **

The graph that is left after step \( i \)

![Graph illustration](image)

After iteration 1

After iteration 2
Maximal Independent Set, Bound on Depth

**Lemma (Degree)**

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

Graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$, $n = |V|

for $i = 1$ to $n$
  if for any earlier neighbor $v_j$ of $v_i$, $S[j] = \text{in}$
  then $S[i] = \text{out}$ else $S[i] = \text{in}$
Maximal Independent Set, Bound on Depth

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

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.
Graph $G = (V, E)$, $S[1, \ldots, n] = \text{unknown}$, $n = |V|

\text{for } i = 1 \text{ to } n
\quad \text{if for any earlier neighbor } v_j \text{ of } v_i, S[j] = \text{in}
\quad \text{then } S[i] = \text{out} \quad \text{else } S[i] = \text{in}

\textbf{Lemma (Degree)}

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

\textbf{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.
Consider increasing sized blocks of the iterations

\[ \log n \quad 2 \log n \quad 4 \log n \ldots \]
Consider increasing sized blocks of the iterations

\[
\begin{align*}
&\log n & 2\log n & 4\log n & \ldots \\
&O(\log n) & O(\log n) & O(\log n)
\end{align*}
\]

w.h.p. no path within a block is greater than \(O(\log n)\)
Consider increasing sized blocks of the iterations:

- \( \log n \)
- \( 2 \log n \)
- \( 4 \log n \)
- \( \ldots \)

By the union bound, the probability of any path of length \( l \) is at most:

\[
\left( \frac{1}{2^i} \right) \left( \frac{2^i \log n}{l} \right) < \left( \frac{1}{2^i} \right) \left( \frac{e2^i \log n}{l} \right) = \left( \frac{e \log n}{l} \right)
\]

Therefore, probability is very small that \( l > 2e \log n \).
Consider increasing sized blocks of the iterations

\[
\log n \quad 2 \log n \quad 4 \log n \quad \ldots
\]

\[O(\log n) \quad O(\log n) \quad O(\log n)\]

**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 \(O(\log n \log d_{\text{max}})\) by picking blocks of size \(2^i \frac{d_{\text{max}}}{n} \log n\).
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., pending iterations in every prefix $0, \ldots, i$ only update $\bigcup_{j \in [i]} l_j$. 

SPAA/PODC 2017

Some Sequential Algorithms are Almost Always Parallel 26 / 35
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 $\bigcup_{j \in [i]} l_j$.

### Example

<table>
<thead>
<tr>
<th>iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>active locations</td>
<td>7</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>for prefix 0,\ldots,5</td>
<td>{1,3,5,7}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>finished</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pending</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SPAA/PODC 2017

Some Sequential Algorithms are Almost Always Parallel 26 / 35
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 $\bigcup_{j \in [i]} l_j$.

**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
### Parallel (Priority PRAM)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth (Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>BST Sort</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Knuth Shuffle</td>
<td>$O(n)$</td>
<td>$O(\log n \log^* n)$</td>
</tr>
<tr>
<td>List Contraction</td>
<td>$O(n)$</td>
<td>$O(\log n \log^* n)$</td>
</tr>
<tr>
<td>2d Linear Programming</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delaunay triangulation</td>
<td>$O(n \log n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
</tbody>
</table>

**All are work efficient.** $\Delta =$ maximum degree
## Implication For Algorithms

### Parallel (Priority PRAM)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Work</th>
<th>Depth (Time)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>$O(</td>
<td>E</td>
</tr>
<tr>
<td>BST Sort</td>
<td>$O(n \log n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Knuth Shuffle</td>
<td>$O(n)$</td>
<td>$O(\log n \log^* n)$</td>
</tr>
<tr>
<td>List Contraction</td>
<td>$O(n)$</td>
<td>$O(\log n \log^* n)$</td>
</tr>
<tr>
<td>2d Linear Programming</td>
<td>$O(n)$</td>
<td>$O(\log n)$</td>
</tr>
<tr>
<td>Delaunay triangulation</td>
<td>$O(n \log n)$</td>
<td>$O(\log^2 n)$</td>
</tr>
</tbody>
</table>

**All are work efficient.** $\Delta = \text{maximum degree}$

### Distributed (CONGEST model)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Rounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIS</td>
<td>$O(\log</td>
</tr>
<tr>
<td>Maximal Matching</td>
<td>$O(\log</td>
</tr>
</tbody>
</table>
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

```c
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
```
Implementation: Speculative For

```c
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

```c
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};
}
```

```c
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.
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

```c
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);```

SPAA/PODC 2017
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);
tseq = best sequential algorithm

\[ t1 = \text{time on one core} \]
\[ t64 = \text{time on all cores} \]
tseq = best sequential algorithm
	t1 = time on one core
	t64 = time on all cores
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.
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.