FROM

MODERN ALGORITHMS WORKSHOP

Parallel Algorithms

Prof. Charles E. Leiserson

Dr. Tao B. Schardl

September 19, 2018

© 2018 Charles E. Leiserson and Tao B. Schardl

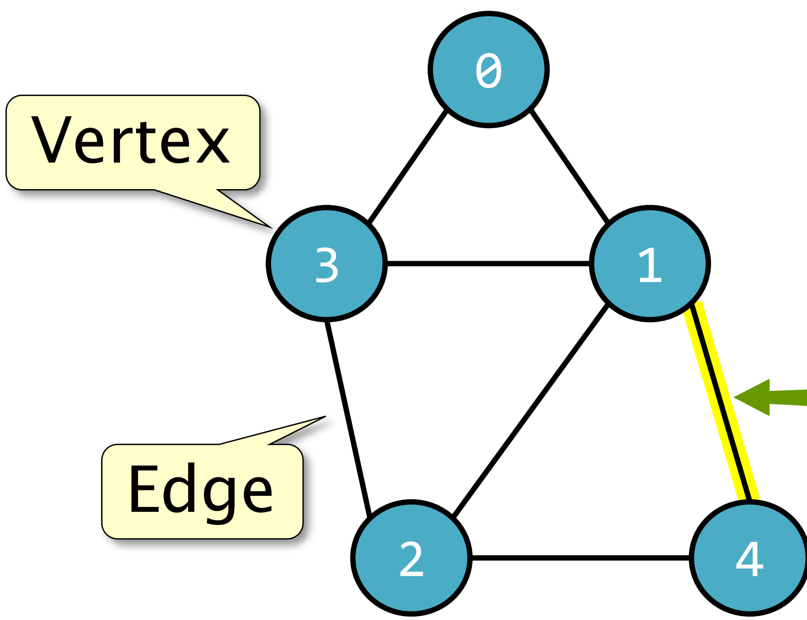
Outline

- Introduction
- Cilk Model
- Detecting Nondeterminism
- What Is Parallelism?
- Scheduling Theory Primer
- *Lunch Break*
- Analysis of Parallel Loops
- Case Study: Matrix Multiplication
- Case Study: Jaccard Similarity
- Post-Moore Software

Case Study: Jaccard Similarity

Graphs

Many problems can be formulated on ***graphs***.
A graph G=(V,E) is a set V of vertices and a set E of edges connecting pairs of vertices.



Adjacency matrix

$$
\begin{array}{c c}
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array} &
\begin{pmatrix}
0 & 1 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 & \boxed{1} \\
0 & 1 & 0 & 1 & 1 \\
1 & 1 & 1 & 0 & 0 \\
0 & \boxed{1} & 1 & 0 & 0
\end{pmatrix} \\
 & \begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array}
\end{array}
$$

A graph can be represented as an ***adjacency matrix***.

© 2018 Charles E. Leiserson and Tao B. Schardl

Problem Statement

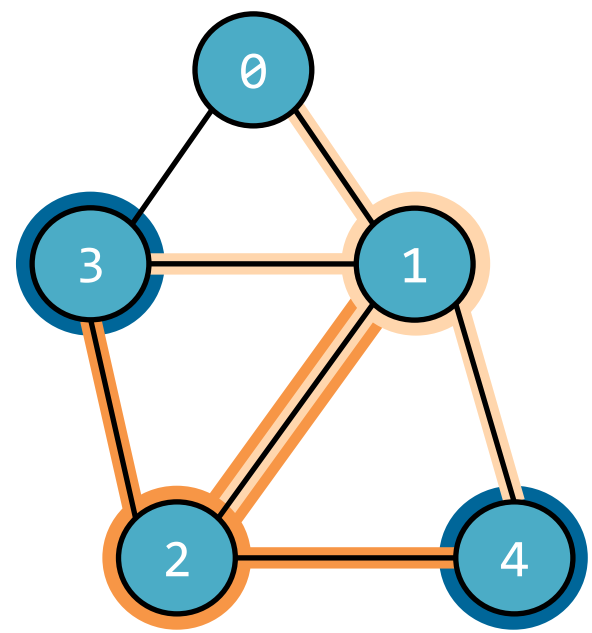
Problem: Given a graph $G=(V,E)$, compute the ***Jaccard similarity*** of every pair of vertices $u,v \in V$, that is,

$$\frac{|Adj[u] \cap Adj[v]|}{|Adj[u] \cup Adj[v]|}$$

where $Adj[u]$ denotes the set of vertices connected to u by an edge.

Jaccard similarities

$$\begin{array}{c}
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \end{array}
\begin{pmatrix}
1 & 1/5 & 2/3 & 1/4 & 1/3 \\
1/5 & 1 & \boxed{2/5} & 2/5 & 1/5 \\
2/3 & \boxed{2/5} & 1 & 1/5 & 2/4 \\
1/4 & 2/5 & 1/5 & 1 & 2/3 \\
1/3 & 1/5 & 2/4 & 2/3 & 1
\end{pmatrix} \\
\begin{array}{ccccc} 0 & 1 & 2 & 3 & 4 \end{array}
\end{array}$$



© 2018 Charles E. Leiserson and Tao B. Schardl

Using Matrix Multiplication

Let A denote the adjacency matrix of graph $G=(V,E)$. Then one can compute the Jaccard-similarity matrix JS as follows:

$\text{Intersection} = A \cdot_{\&} A$

Matrix multiply using bitwise AND

$\text{Union} = A \cdot_{|} A$

Matrix multiply using bitwise OR

$JS = \text{Intersection} \div_{\text{el}} \text{Union}$

Element-wise division

$T_1 = \Theta(M_1(A)) = \Theta(V^3)$

Work of matrix multiplication

Can we do better for ***sparse graphs***, where $|E| \ll |V|^2$?

Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes. *"The fastest way to compute is not to compute at all."*

Example: Matrix–vector multiplication

$$y = \begin{pmatrix} 3 & 0 & 0 & 0 & 1 & 0 \\ 0 & 4 & 1 & 0 & 5 & 9 \\ 0 & 0 & 0 & 2 & 0 & 6 \\ 5 & 0 & 0 & 3 & 0 & 0 \\ 5 & 0 & 0 & 0 & 8 & 0 \\ 0 & 0 & 0 & 9 & 7 & 0 \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 2 \\ 8 \\ 5 \\ 7 \end{pmatrix}$$

Dense matrix–vector multiplication performs $n^2 = 36$ scalar multiplies, but only 14 entries are nonzero.

© 2008–2018 by the MIT 6.172 Lecturers

Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes. *"The fastest way to compute is not to compute at all."*

Example: Matrix–vector multiplication

$$y = \begin{pmatrix} 3 & & & & 1 & \\ & 4 & 1 & & 5 & 9 \\ & & & 2 & & 6 \\ 5 & & & 3 & & \\ 5 & & & & 8 & \\ & & & 9 & 7 & \end{pmatrix} \begin{pmatrix} 1 \\ 4 \\ 2 \\ 8 \\ 5 \\ 7 \end{pmatrix}$$

Dense matrix–vector multiplication performs $n^2 = 36$ scalar multiplies, but only 14 entries are nonzero.

Sparsity (2)

Compressed Sparse Row (CSR)

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
rows:	0	2	6	8	10	11	14							
cols:	0	4	1	2	4	5	3	5	0	3	0	4	3	4
vals:	3	1	4	1	5	9	2	6	5	3	5	8	9	7

$$
\begin{array}{c c}
\begin{array}{c} 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} &
\begin{pmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 8 & 9 & 7
\end{pmatrix} \\
& \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array}
\end{array}
$$

$n = 6$

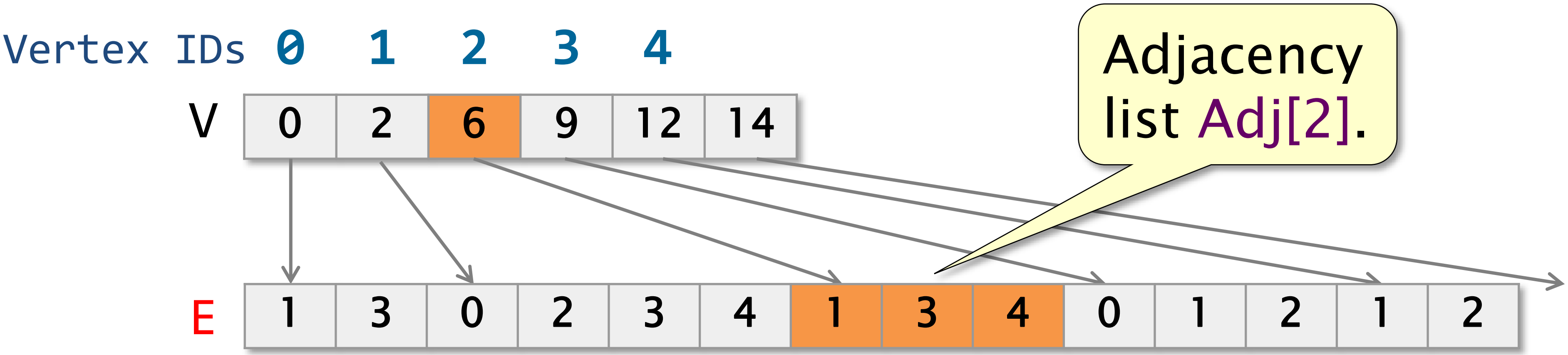
$nnz = 14$

Storage is $O(n+nnz)$ instead of n^2

© 2008–2018 by the MIT 6.172 Lecturers

Sparse Graph Representation

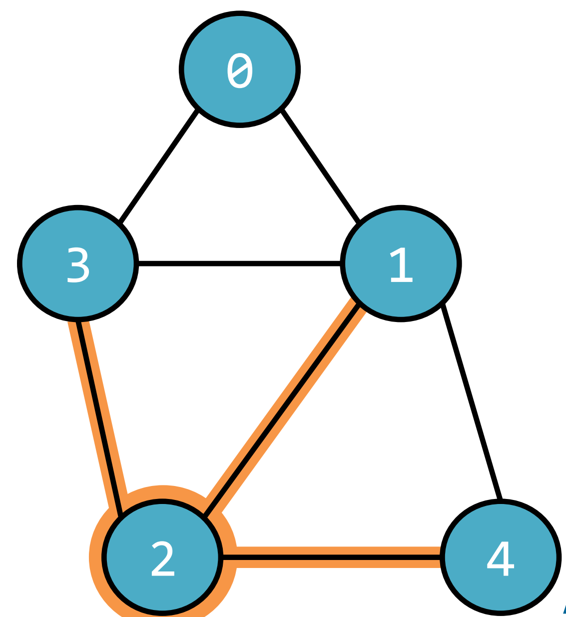
Storing a sparse graph $G=(V,E)$ using ***compressed sparse rows (CSR)***.



Can run many graph algorithms efficiently on this representation, e.g., breadth-first search, PageRank.

Storage is $\Theta(V+E)$ instead of $|V|^2$.

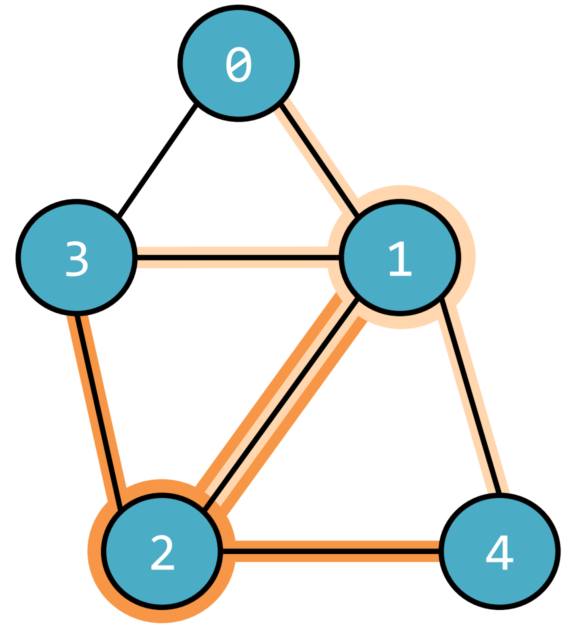
Adjacency lists are typically sorted.



© 2018 Charles E. Leiserson and Tao B. Schardl

Key Computation: List Intersection

Key to computing the Jaccard similarity of a pair of vertices $u,v \in V$ is to compute $|Adj[u] \cap Adj[v]|$.



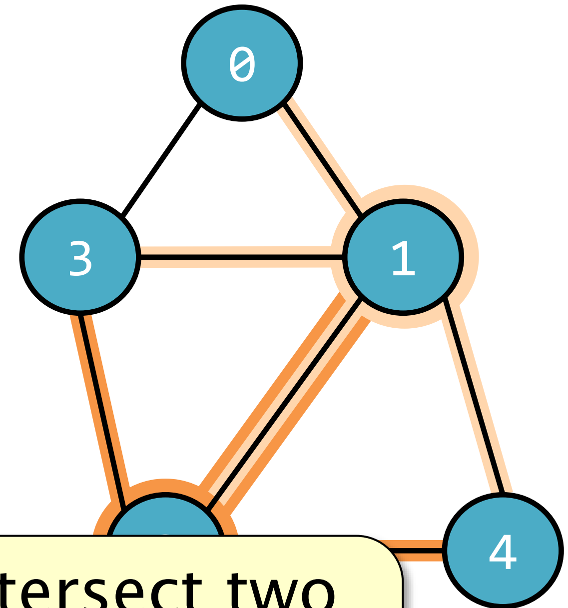
Vertex IDs	0	1	2	3	4	
V	0	2	6	9	12	14

E	1	3	0	2	3	4	1	3	4	0	1	2	1	2

Using CSR, computing $|Adj[u] \cap Adj[v]|$ involves computing the size of the intersection of two sorted lists of integers.

Intersecting Two Adjacency Lists

```
int intersect(const int *AdjA, int na,
              const int *AdjB, int nb) {
  int intersection = 0;
  while (na>0 && nb>0) {
    if (*AdjA == *AdjB) {
      intersection++;
      AdjA++; na--; AdjB++; nb--;
    } else if (*AdjA < *AdjB) {
      AdjA++; na--;
    } else { // *AdjB < *AdjA
      AdjB++; nb--;
    }
  }
  return intersection;
}
```



Time to intersect two lists Adj[u] and Adj[v] is Θ(Adj[u] + Adj[v]).

intersection 2

AdjA	0	2	3	4
AdjB	1	3	4	

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

© 2018 Charles E. Leiserson and Tao B. Schardl

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

Output matrix of Jaccard-similarity values.

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[u
                  E[V[vi]], V[vi+1] - V[v
}
```

Vertex array in CSR, storing offsets into edge array.

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

Number of vertices.

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

Edge array.

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

Iterate over all pairs of vertices.

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V[ui],
                  E[V[vi]], V[vi+1] - V[vi]);
}
```

Compute the intersection of the adjacency lists and store the result.

© 2018 Charles E. Leiserson and Tao B. Schardl

Analysis of Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] - V
                  E[V[vi]], V[vi+1] - V[vi
}
```

Degree of vertex u*:*
$d(u) = |Adj[u]|$

Work: $T_1(G) = \sum_{u\in V}\sum_{v\in V}(d(u)+d(v))$

Serial Jaccard Similarity

For simplicity, let us focus on computing the intersection of each pair of adjacency lists.

```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] =
        intersect(E[V[ui]], V[ui+1] -
                  E[V[vi]], V[vi+1] -
}
```

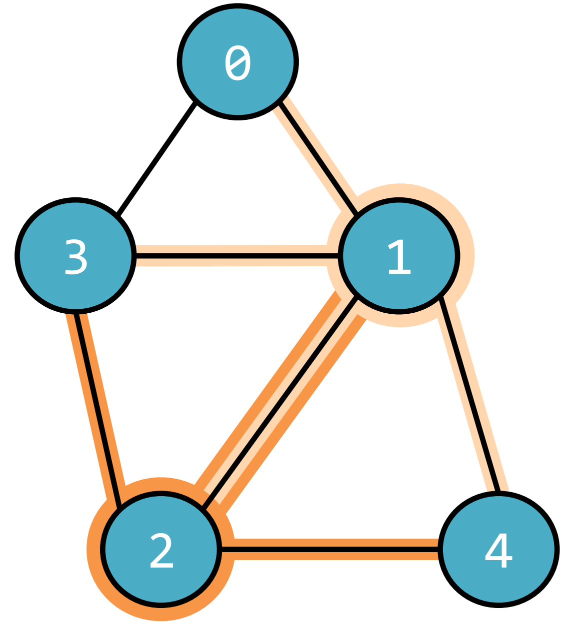
Handshaking Lemma:
$\Sigma_{u \in V} d(u) = 2|E|$

Work: $T_1(G) = \Sigma_{u \in V} \Sigma_{v \in V} (d(u) + d(v))$
$= \Sigma_{u \in V} \Sigma_{v \in V} d(v) + \Sigma_{v \in V} \Sigma_{u \in V} d(u)$
$= 2|V||E| + 2|V||E|$
$= \Theta(VE)$

© 2018 Charles E. Leiserson and Tao B. Schardl

Exploiting Symmetry

In an undirected graph, we have $|\text{Adj}[u] \cap \text{Adj}[v]| = |\text{Adj}[v] \cap \text{Adj}[u]|$.
Hence, the intersection of each pair of adjacency lists can be computed just once.




```
void jaccard(int *JS,
             const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi <= ui; ++vi)
      JS[ui*nv+vi] = JS[vi*nv+ui] =
        intersect(E[V[ui]], V[ui+1]-V[ui], E[V[vi]], V[vi+1]-V[vi]);
}
```

Work: $T_1(G) = \Theta(VE)$

© 2018 Charles E. Leiserson and Tao B. Schardl

Parallel Jaccard Similarity V.1

```
void p1_jaccard(int *JS,
                const int *V, int nv, const int *E) {
  cilk_for (int ui = 0; ui < nv; ++ui)
    for (int vi = 0; vi <= ui; ++vi)
      JS[ui*nv+vi] = JS[vi*nv+ui] =
        intersect(E[V[ui]], V[ui+1]-V[ui], E[V[vi]], V[vi+1]-V[vi]);
}
```

Parallel outer loop.

Span: $T_\infty(G) = \lg|V| + \max_{u \in V}\{\sum_{v \in V}(d(u)+d(v))\}$

$= \lg|V| + \max_{u \in V}\{\sum_{v \in V} d(u) + \sum_{v \in V} d(v)\}$

$= \lg|V| + \max_{u \in V}\{|V|d(u)+2|E|\}$

$= \Theta(V\Delta + E)$, where $\Delta = \max_{u \in V}\{d(u)\}$

$= \Theta(V\Delta)$

Parallelism of Jaccard Similarity V.1

Work: $T_1(n) = \Theta(VE)$

Span: $T_\infty(n) = \Theta(V\Delta + E)$

Parallelism:

$$\frac{T_1(n)}{T_\infty(n)} = \Theta(VE/(V\Delta)) = \Theta(E/\Delta)$$

Parallel Jaccard Similarity V.2

```
void p2_jaccard(int *JS,
                const int *V, int nv, const int *E) {
  cilk_for (int ui = 0; ui < nv; ++ui)
    cilk_for (int vi = 0; vi <= ui; ++vi)
      JS[ui*nv+vi] = JS[vi*nv+ui] =
        intersect(E[V[ui]], V[ui+1]-V[ui], E[V[vi]], V[vi+1]-V[vi]);
}
```

Both loops are parallel.

Span: $T_\infty(G) = 2\lg|V| + \max_{u \in V}\left\{\max_{v \in V}\{d(u)+d(v)\}\right\}$

$= \Theta(\lg V + \Delta)$

Parallelism of Jaccard Similarity V.2

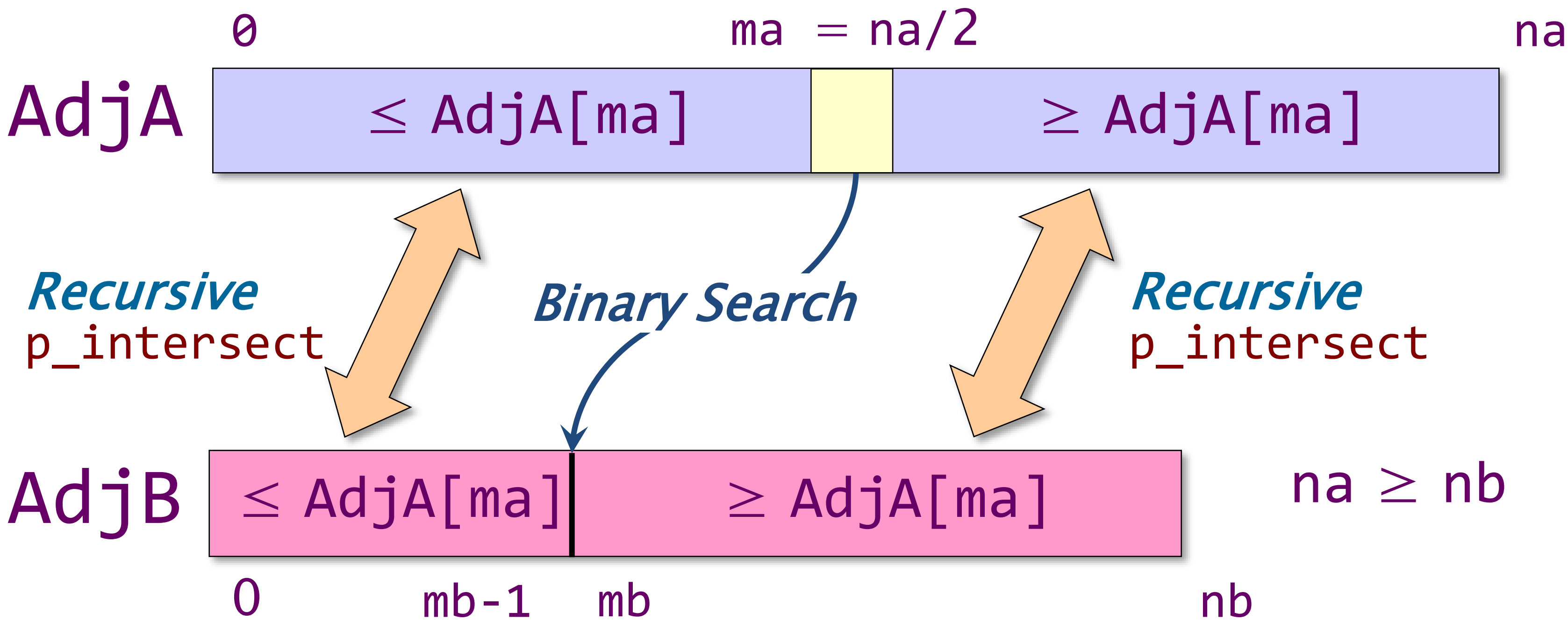
Work: $T_1(n) = \Theta(VE)$

Span: $T_\infty(n) = \Theta(\lg V + \Delta)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(VE/(\lg V + \Delta))$

$= \Omega(E)$ worst case

Parallel Intersect



KEY IDEA: If the total number of elements to be intersected in the two arrays is $n = na + nb$, the total number of elements in the larger of the two recursive merges is at most $(3/4)n$.

© 2018 Charles E. Leiserson and Tao B. Schardl

Parallel Intersect Code

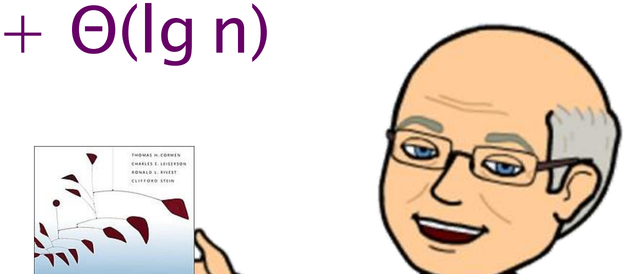
```
int p_intersect(const int *AdjA, int na, const int *AdjB, int nb) {
  if (na < nb) {
    return p_intersect(AdjB, nb, AdjA, na);
  } else if (na < THRESHOLD) {
    return intersect(AdjA, na, AdjB, nb);
  } else {
    int ma = na/2;
    int mb = binary_search(AdjA[ma], AdjB, nb);
    int intersection_l, intersection_r;
    intersection_l = cilk_spawn p_intersect(AdjA, ma, AdjB, mb);
    intersection_r = p_intersect(AdjA+ma, na-ma, AdjB+mb, nb-mb);
    cilk_sync;
    return intersection_l + intersection_r;
  }
}
```

© 2018 Charles E. Leiserson and Tao B. Schardl

Span of Parallel Intersect

```
int p_intersect(const int *AdjA, int na, const int *AdjB, int nb) {
  if (na < nb) {
    return p_intersect(AdjB, nb, AdjA, na);
  } else if (na < THRESHOLD) {
    return intersect(AdjA, na, AdjB, nb);
  } else {
    int ma = na/2;
    int mb = binary_search(AdjA[ma], AdjB, nb);
    int intersection_l, intersection_r;
    intersection_l = cilk_spawn p_intersect(AdjA, ma, AdjB, mb);
    intersection_r = p_intersect(AdjA+ma, na-ma, AdjB+mb, nb-mb);
    cilk_sync;
    return intersection_l + intersection_r;
  }
}
```

Span: $$T_\infty(n) = T_\infty(3n/4) + \Theta(\lg n) = \Theta(\lg^2 n)$$



© 2018 Charles E. Leiserson and Tao B. Schardl

Work of Parallel Intersect

```
int p_intersect(const int *AdjA, int na, const int *AdjB, int nb) {
  if (na < nb) {
    return p_intersect(AdjB, nb, AdjA, na);
  } else if (na < THRESHOLD) {
    return intersect(AdjA, na, AdjB, nb);
  } else {
    int ma = na/2;
    int mb = binary_search(AdjA[ma], AdjB, nb);
    int intersection_l, intersection_r;
    intersection_l = cilk_spawn p_intersect(AdjA, ma, AdjB, mb);
    intersection_r = p_intersect(AdjA+ma, na-ma, AdjB+mb, nb-mb);
    cilk_sync;
    return intersection_l + intersection_r;
  }
}
```

Work: $T_1(n) = T_1(\alpha n) + T_1((1-\alpha)n) + \Theta(\lg n)$, where $1/4 \le \alpha \le 3/4$.

HAIRY!

Solution: $T_1(n) = \Theta(n)$.

© 2018 Charles E. Leiserson and Tao B. Schardl

Parallelism of Parallel Intersect

Work: $T_1(n) = \Theta(n)$

Span: $T_\infty(n) = \Theta(\lg^2 n)$

Parallelism: $\dfrac{T_1(n)}{T_\infty(n)} = \Theta(n/\lg^2 n)$

© 2018 Charles E. Leiserson and Tao B. Schardl

Parallel Jaccard Similarity V.3

```
void p3_jaccard(int *JS,
                const int *V, int nv, const int *E) {
  cilk_for (int ui = 0; ui < nv; ++ui)
    cilk_for (int vi = 0; vi < nv; ++vi)
      JS[ui*nv+vi] = JS[vi*nv+ui] =
        p_intersect(E[V[ui]], V[ui+1]-V[ui],
                    E[V[vi]], V[vi+1]-V[vi]);
}
```

Span:

$$\begin{aligned} T_\infty(G) &= 2\lg|V| + \max_{u \in V}\left\{\max_{v \in V}\left\{\lg^2(d(u)+d(v))\right\}\right\} \\ &\le 2\lg|V| + \lg^2(2\Delta) \\ &= \Theta(\lg V + \lg^2\Delta) \end{aligned}$$

© 2018 Charles E. Leiserson and Tao B. Schardl

Parallelism of Jaccard Similarity V.3

Work: $T_1(n) = \Theta(VE)$

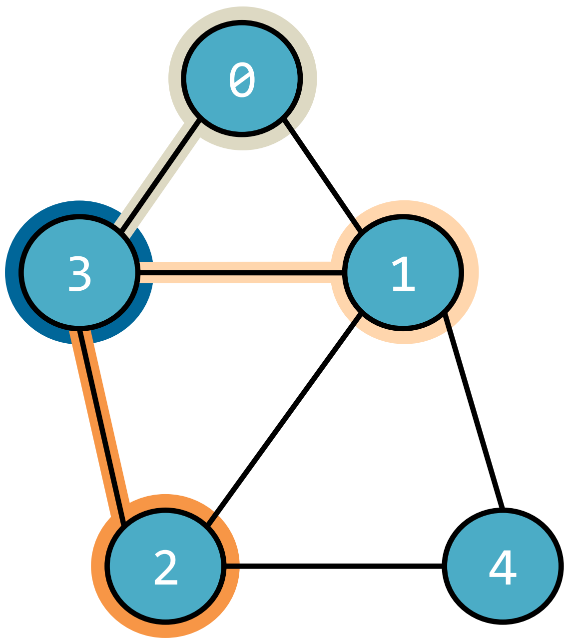
Span: $T_\infty(n) = \Theta(\lg V + \lg^2\Delta)$

Parallelism: $$\frac{T_1(n)}{T_\infty(n)} = \Theta(VE/(\lg V + \lg^2\Delta))$$

© 2018 Charles E. Leiserson and Tao B. Schardl

Another Approach

Observation: For any vertex u, for any pair of vertices v,w ∈ Adj[u], vertex u contributes 1 to |Adj[v] ∩ Adj[w]|.



Example: Vertex 3 increases by 1 the size of the intersections:

- |Adj[0] ∩ Adj[1]|
- |Adj[0] ∩ Adj[2]|
- |Adj[1] ∩ Adj[2]|

Idea: For each vertex u, iterate over all pairs of vertices in Adj[u] and increment the intersection size for the pair.

© 2018 Charles E. Leiserson and Tao B. Schardl

Push Algorithm for Jaccard Similarity

```
void push_jaccard(int *JS,
                  const int *V, int nv, const int *E) {
  for (int ui = 0; ui < nv; ++ui) {
    for (int vi = V[ui]; vi < V[ui+1]; ++vi) {
      for (int wi = V[ui]; wi < V[ui+1]; ++wi) {
        int v = E[vi], w = E[wi];
        JS[v*nv + w]++;
      }
    }
  }
}
```

Work:

$$
\begin{aligned}
T_1(G) &= \textstyle\sum_{u\in V}\sum_{v\in Adj[u]}\sum_{w\in Adj[u]}\Theta(1) \\
&= \textstyle\sum_{u\in V} d(u)^2 \\
&\le \Delta \textstyle\sum_{u\in V} d(u) \\
&= O(\Delta E)
\end{aligned}
$$

© 2018 Charles E. Leiserson and Tao B. Schardl

Hybrid Algorithm for Jaccard Similarity

Idea: Process the low-degree and high-degree vertices separately.

- Use the push algorithm to handle vertices with degree less than $\sqrt{E}$.
- Use the $\Theta(VE)$ algorithm to handle the remaining high-degree vertices.

Work: $T(G) = O(V + \min\{\Delta E, E^{3/2}\})$

- Partitioning the vertices requires $\Theta(V)$ work.
- Processing low-degree vertices requires $O(\min\{\Delta E, E^{3/2}\})$ work.
- At most $O(\sqrt{E})$ vertices can have high degree, i.e., degree at least $\sqrt{E}$.
- Processing high-degree vertices requires $\Theta(E^{3/2})$ work.

© 2018 Charles E. Leiserson and Tao B. Schardl

What About Parallelism?

Take-home puzzles

- How do you parallelize the Push algorithm for Jaccard similarity?
- How do you parallelize the Hybrid algorithm?
- How well do these parallel algorithms work in practice?

© 2018 Charles E. Leiserson and Tao B. Schardl