Euler Tours and Their Applications

Chris Moultrie
CSc 8530
Topics Covered

- History
- Terminology
- Sequential Algorithm
- Parallel Algorithm
- BSP/CGM Algorithm
History

Started with the famous Konigsberg problem. In the city of Kongisberg its parks were built on either side of a river. In the river are two islands and seven bridges. The puzzle asked if it was possible to walk through the park crossing each bridge only once.

Euler solved this $O(n!)$ problem proving that no, it was not possible.
Terminology

**Graph** G=(V,E) is a set of vertices V connected by a set of edges E.

**Connected**: two vertexes are connected if G contains a path from u to v :: Example (A,D)

**Undirected**: the edges have no orientation

**Path** in G is an ordered list of edges expressed:

(\(v_i,v_j\)),(\(v_j,v_k\)),(\(v_k,v_l\)) :: Example: (D,A),(A,B),(B,C)
Terminology

- **Directed**: every edge has an orientation

- **Cycle** in G is a path that begins and ends at the same vertex
  - Example: (A,B), (B,C), (C,A)
Terminology

A Tree is an undirected graph that is connected and contains no cycles.

A tree with $n$ vertices has exactly: $(n-1)$ edges.
**Terminology**

**Euler Tour** is a cycle that includes every edge only once and returns to its origin.

Example: (A,B),(B,C),(C,D),(D,E),(E,F),(F,D),(D,A)
Necessary and Sufficient Conditions

- An undirected graph must be connected and every vertex must have an even degree.
- A directed graph must be connected and every vertex must have equal in and out degrees.
- A directed tree is denoted $DT$ and an Eulerian Tour on $DT$ by $ET$. 
1. While G has degree > 1
2. Start at a vertex and walk the graph until you complete a cycle
3. Remove the cycle from the graph
4. Choose a vertex in both the remaining graph and the removed cycle
5. Walk a cycle from the new vertex
6. Repeat until no more edges
Constructing the Eulerian Tour

To build the tour from the first traversal of the graph, one more traversal is needed. The cycles are joined in a greedy fashion.

The path becomes:
(A, B), (B, C), (C, D), (D, E), (E, G), (G, H), (H, I), (I, J), (J, H), (H, D), (D, A)
Sequential algorithm takes $O(E)$ time because it must traverse the graph twice. Once to find all the cycles and a second time to merge them.

$2 * E = O(E)$
A Parallel Algorithm

Constructing an ET on a DT with n vertices is possible in parallel.

DT is stored in n linked lists with each node - each edge is an out-path from that node.

Each node $ij$ in for vertex $v_i$ contains 2 fields:

- **edge** - the edge $(v_i, v_j)$
- **next** - pointer to next node in list

Result: Sequentially list all elements in one linked list with each $(v_i, v_j)$ followed by $(v_j, v_k)$.
Linked list of a DT

Head (v₁) → (V1,V3) → (V1,V4) → (V1,V5) → Head (v₂) → (V2,V7) → Head (v₃) → (V3,V1) → Head (v₄) → (V4,V1) → Head (v₅) → (V5,V1) → (V5,V6) → (V5,V7) → Head (v₆) → (V6,V5) → Head (v₇) → (V7,V2) → (V7,V5)
PRAM Algorithm

$n-1$ processors each in charge of $P_{ij}$ ($i < j$) for two edges $(v_i, v_i)$ and $(v_i, v_i)$

$P_{ij}$ determine position in ET of 2 nodes holding $(v_i, v_i)$ and $(v_i, v_i)$ denoted by $ij$ and $ji$

To determine successor of $(v_i, v_i)$:

- If linked list of $v_i$ edge $(v_i,v_i)$ followed by $(v_i,v_k)$ then successor of $(v_i,v_i)$ in ET is $(v_i,v_k)$
- Else successor of $(v_i,v_i)$ in ET is the first edge in the linked list for $v_i$
Successor of \((v_i, v_j)\)

if \(\text{next}(ji) = jk\)
then \(\text{succ}(ij) \leftarrow jk\)
else \(\text{succ}(ij) \leftarrow \text{head}(v_i)\)
end if

Successor of \((v_i, v_j)\)

if \(\text{next}(ij) = im\)
then \(\text{succ}(ji) = im\)
else \(\text{succ}(ji) \leftarrow \text{head}(v_i)\)
end if

**Analysis of PRAM Algorithm**

Each successor is found in constant time, therefore with \(n-1\) processors our time is \(O(1)\).
Euler Tour PRAM Result

Head (v1)
(A, B)

Head (v2)
(B, C)

Head (v3)
(C, D)

Head (v4)
(D, G)

Head (v5)
(G, E)

Head (v6)
(E, F)

Head (v7)
(F, G)
Depth-First Traversal

Designate a special vertex $v_1$ as the root

Method for Depth-First Traversal:

- Visit the root

- Visit each of the subtrees of the root (recursively) in depth-first order.

This is also known as pre-order traversal
Depth-First Traversal

Depth first traversal of this tree: B, A, F, D, E, G, C, H

An ET of a rooted DT can be made to correspond to a DFT by choosing an edge “leaving” the root as the first edge of the ET.
With some changes, the linked list representing the tree can be made into the form of a linked list described in 6.2.

Each node $ij$ of ET is in the format:

- info field stores the edge $(v_i, v_j)$
- val field defined when required

Functions from 6.2 such as prefix computation and list ranking can be performed on ET.
If we assume all nodes initialized with $\text{val}(i) \leftarrow 1$ then prefix sum will give us the node's position (denoted $\text{pos}(v_i, v_j)$) stored in $\text{val}(i)$.

This, of course, will take us $O(\log n)$ time with $O(n)$ processors.
**Basic Tree Computations**

(vᵢ, vᵢ) is an **advance edge** of DT if pos(vᵢ, vᵢ) < pos(vᵢ, vᵢ), else it is called a **retreat edge**

Finding a parent is easy:

- parent(vᵢ) ← vᵢ for each advance edge (vᵢ, vᵢ)
- if vᵣ is the root, we can set parent(vᵣ) ← nil
Enumerating Descendants

For each vertex $v_i$ (except root) can be determined from:

$$\text{des}(v_i) \leftarrow (\text{pos}(v_i, \text{parent}(v_i)) - 1 - \text{pos}(\text{parent}(v_i), v_i))/2 + 1$$

$\text{des}(v_5) = ?$
$\text{pos}(v_5, \text{parent}(v_5)) = 12$
$\text{pos}(\text{parent}(v_5), v_5) = 5$
$\text{des}(v_5) = (12-1-5)/2 + 1 = 4$
An Euler tour represents a preorder traversal of DT

The preorder number of a vertex $v_i$ is the number of advance edges traversed before reaching $v_i$ for the first time plus one.

First assign 1 to the $val(i)$ field in each vertex if $(v_i, v_i)$ is an advance edge, else assign 0, then perform prefix sum.

If $(v_i, v_i)$ is an advance edge, then

$$ preorder(v_i) \leftarrow val(ij) + 1 $$
Numbering the Vertices

Postorder Traversal (bottom-up)

1. visit each of the subtrees of the root (recursively) in postorder.
2. Visit the root

A vertex is only visited after all of its descendants are visited

This can be computed as follows:

- Assign values to val fields of ET, if \((v_i, v_j)\) is a retreat edge, then \(\text{val}(ij) = 1\); else 0
- Perform a prefix sums computation over the val fields
- \(\text{postorder}(v_i) \leftarrow \text{val}(ij)\) for a retreat edge \((v_i, v_j)\)
Example of Numbering Edges

\[(v_1,v_3) \rightarrow (v_3,v_1) \rightarrow (v_1,v_4) \rightarrow (v_4,v_1)\]
\[(v_5,v_7) \rightarrow (v_5,v_6) \rightarrow (v_6,v_5) \rightarrow (v_1,v_5)\]
\[(v_7,v_2) \rightarrow (v_2,v_7) \rightarrow (v_7,v_5) \rightarrow (v_5,v_1)\]

<table>
<thead>
<tr>
<th>edge ((v_i,v_j))</th>
<th>((v_1,v_3))</th>
<th>((v_3,v_1))</th>
<th>((v_1,v_4))</th>
<th>((v_4,v_1))</th>
<th>((v_1,v_5))</th>
<th>((v_5,v_6))</th>
<th>((v_6,v_5))</th>
<th>((v_5,v_7))</th>
<th>((v_7,v_2))</th>
<th>((v_2,v_7))</th>
<th>((v_7,v_5))</th>
<th>((v_5,v_1))</th>
</tr>
</thead>
<tbody>
<tr>
<td>val</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>prefix sum</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>post-order</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Evaluating Binary Relations

Two binary relations between vertices are “is a descendant of” and “is an ancestor of”

$v_i$ is an ancestor of $v_j$ iff both of these are true:

- $\text{preorder}(v_i) \leq \text{preorder}(v_j)$
- $\text{preorder}(v_i) < \text{preorder}(v_i) + \text{des}(v_i)$
**Example of Binary Relation**

\[
\begin{align*}
\text{\textbf{v}_1 \text{ is an ancestor of } \textbf{v}_2 \text{ since:}} \\
\text{preorder}(v_1) &= 1, \text{preorder}(v_2) = 7, \quad \text{des}(v_1) = 7 \\
1 &\leq 7 \text{ and } 7 < 1 + 7 \\
\text{\textbf{v}_3 \text{ is not an ancestor of } \textbf{v}_5 \text{ since:}} \\
\text{preorder}(v_3) &= 2, \text{preorder}(v_5) = 4, \text{des}(v_3) = 1 \\
\text{preorder}(v_5) &< \text{preorder}(v_3) + \text{des}(v_3) \Leftarrow \text{Not True!}
\end{align*}
\]
Let $T$ be the rooted tree with $n$ vertices

Assuming that $\text{pos}(v_i, v_j)$ has already been computed for all edges of $ET$ then:

- $\text{parent}(v_j)$ and $\text{des}(v_j)$ can be computed in $O(1)$ time with $O(n)$ processors for all vertices of $T$

However, $\text{preorder}(v_i)$, $\text{postorder}(v_i)$ and $\text{level}(v_i)$ for all $v_i$ in $T$ require prefix computation, and therefore $O(\log n)$ time and $O(n)$ processors.
Computing Minima

Every vertex in T stores an arbitrary number

Find, for every vertex $v_i$ of T the min/max of all numbers in the subtree rooted at $v_i$

Can be solved with pointer jumping, but can also be solved with an Euler Tour

Consider the sequence $\{s_k, s_{k+1}, ..., s_l\}$ where $k=\text{preorder}(v_i)$ and $l=k+\text{des}(v_i) -1$
Computing Minima

Preorder Traversal:
$V_1, V_2, V_5, V_7, V_6, V_8, V_3, V_4$

$\{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\} = \{10, 5, 6, 7, 3, 4, 2, 1\}$

Numbers in a subtree rooted at $v_2$ appear in: $s_2, s_3, s_4, s_5, s_6$ since $\text{preorder}(v_2) = 2, \text{des}(v_2) = 5, (2+5-1)=6$
Parallel Algorithm for Minima

Given tree $T$ rooted at $v_i$ (for all $v_i$ in $T$)

1. Convert $T$ into a DT, compute $ET$

2. Minimum number stored in $T$ rooted at $v_i$ is: 
   $\min\{s_k, s_{k+1}, ..., s_l\}$, where $k=\text{preorder}(v_i)$ and $l=k+\text{des}(v_i) - 1$

There are $n$ pairs $(k,l)$ and therefore $n$ subsequences of $S$ which is an instance of an Interval Minima Problem
3. Compute the minima (assume $n$ is a power of 2)

Build a complete binary tree $T'$ whose leaves are $s_1, s_2, ..., s_n$

Each internal (non-leaf) vertex $x$ contains 2 subsequences:

- Prefix Minima (PM) of the values in the leaves, rooted at $x$
- Suffix Minima (SM) of the values in the leaves, rooted at $x$
Interval Minima Example

PM = 10, 5, 5, 5, 3, 3, 2, 1
SM = 1, 1, 1, 1, 1, 1, 1, 1

PM = 10, 5, 5, 5
SM = 5, 5, 6, 7

PM = 3, 3, 2, 1
SM = 1, 1, 1, 1

PM = 10, 5
SM = 5, 5

PM = 6, 6
SM = 6, 7

PM = 3, 3
SM = 3, 4

PM = 2, 1
SM = 1, 1

10  5
6  7
3  4
2  1
Interval Minima Calculations

Constructing the PM array at each vertex in $T'$

1. Concatenate both PM arrays of children

2. Let $u$ be the last element in the first half of the resulting array PM, each $v$ in the second half is replaced with $\min(u,v)$

Constructing the SM array at each vertex in $T'$

1. Concatenate both SM arrays of children

2. Let $u$ be the first element in the second half of the resulting array SM, each $v$ in the first half is replaced with $\min(u,v)$
For the tree $T'$ let $k=2$ and $l=6$ so $\{s_2, s_3, s_4, s_5, s_6\} = \{5, 6, 7, 3, 4\}$.

Let $w$ be the Lowest Common Ancestor in $T'$ of $k$ and $l$ with $x$ and $y$ as the left and right children of $w$.

$\min \{s_2, s_3, s_4, s_5, s_6\}$ is:

- The suffix minimum corresponding to $k$ in the SM array of $x$ is (5)
- The prefix minimum corresponding to $l$ in the PM array of $y$ is (3)
- $\min(5, 3) = 3$

Since each internal vertex of $T'$ "knows" the range of leaves in its subtree we can begin at the root and descend the tree until the furthest vertex from the root is reached, whose subtree contains the leaves $i, i+1, \ldots, j$ where $i \leq k$ and $l \leq k$.
Analysis of Interval Minima

On a PRAM model with \( n \) processors, the preorder numbers of all vertices in \( T \) can be found in \( O(\log n) \) time as shown earlier.

\( T' \) has \( n \) leaves and hence \( \log n \) levels

In each level \( PM \) and \( SM \) are calculated in \( O(1) \) time using \( n \) processors

\( PM \) or \( SM \) of size \( m \) needs \( m/2 \) processors to copy left child’s \( m/2 \) elements and \( m/2 \) to copy right child’s elements. There is 1 comparison and update involving the last \( m/2 \) elements of \( PM \) or \( SM \)

For each \([k, l]\), 1 processor can find LCA \( \omega \) in \( O(\log n) \) beginning from the root

One comparison between elements of \( L \) and \( R \) children of \( \omega \) is done in constant time.

Time: \( O(\log n) \)  Processors: \( O(n) \)  Cost: \( O(n \log n) \)
The BSP/CGM Model

Bulk Synchronous Parallel/Course-Grained Multicomputer (proposed in [2], BSP model first proposed in [3])

Course Grained - assumption that the problem size is considerably larger than the number of processors

CGM Machine - CGM(n,p) consists of a set of p processors, each with local memory of size O(n/p)

Each processor is connected to a router - can send messages to any other processor

CGM Algorithms - alternating sequence of computation rounds and communication rounds

Communication Round - each processor exchanges at most O(n/p) data with other processors

Effort to reduce communication - in reduction of the number of communication rounds

One of the techniques - reduce the input set s.t. it can be moved to a single processor and then solved sequentially in that processor
Valiant (see [3]) proposed the Bulk Synchronous Parallel (BSP) model as a possible model for parallel computing. He refers to BSP as a “bridging” model, being applicable to both system and algorithm design. The model allows hardware and software design to proceed independently but ensures compatibility between parallel computers and parallel programs.
Graph and Edge Distribution

The input graph $G$

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>(1,5)(2,1)(1,3)(2,6)(3,4)(6,5)(4,1)(7,2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>(6,3)(4,6)(5,4)(3,2)(5,7)(8,7)(14,7)(7,9)</td>
</tr>
<tr>
<td>$P_2$</td>
<td>(8,13)(10,8)(7,13)(9,12)(11,9)(13,9)(13,11)(9,10)</td>
</tr>
<tr>
<td>$P_3$</td>
<td>(12,13)(9,8)(13,14)(16,14)(14,15)(15,16)</td>
</tr>
</tbody>
</table>
BSP/CGM Euler Tour

**Idea:** reducing the input size (Cáceres et al. [4]), main steps based on algorithm of Atallah-Vishkin [5] for the PRAM model

**Input:** (1) \( p \) processors \( p_0, p_1, \ldots, p_{p-1} \)

(2) Eulerian graph \( G = (V, E) \), list of edges stored in array \( edge \), \( edge \) distributed among \( p \) processors, each processor: \( m/p \) edges, \( n/p \) vertices

**Output:** \( succ(i) \) contains successor of edge \( edge(i) \) in the obtained ET

**Step 1:** Obtain an Euler partition \( C = C_1, C_2, \ldots, C_k \) of \( G \)

**Step 2:** Construct the auxiliary bipartite graph that identifies vertices through which pass more than one cycle of \( C \)

**Step 3:** Construct a spanning tree of the bipartite graph and compute the strut of the spanning tree

**Step 4:** Use the stitching operation to perform the union of two or more cycles that pass through a same vertex
Step 1

**Step 1: Obtain Euler Partition** \( C=C_1, C_2, \ldots, C_k \) of \( G \)

Sort the edges by the destination vertices

<table>
<thead>
<tr>
<th>Array</th>
<th>Edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>(2,1)(4,1)(3,2)(7,2)(1,3)(6,3)(3,4)(5,4)</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>(1,5)(6,5)(2,6)(4,6)(5,7)(8,7)(14,7)(9,8)</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>(10,8)(7,9)(11,9)(13,9)(9,10)(13,11)(9,12)(7,13)</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>(8,13)(12,13)(13,14)(16,14)(14,15)(15,16)</td>
</tr>
</tbody>
</table>

Sort the edges by origin vertices

<table>
<thead>
<tr>
<th>Array</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>(1,3) (1,5) (2,1) (2,6) (3,2) (3,4) (4,1) (4,6)</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>(1,5)(6,5)(2,6)(4,6)(5,7)(8,7)(14,7)(9,8)</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>(10,8)(7,9)(11,9)(13,9)(9,10)(13,11)(9,12)(7,13)</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>(8,13) 12,13) (13,14) (16,14) (14,15) (15,16)</td>
</tr>
</tbody>
</table>

Together, Arrays *Edge* and *Succ* form an Euler Partition (a set of disjoint cycles)
Cycles in $G$

$G$ composed of 6 cycles:

$C_1 = \{(2,1),(1,3),(3,2)\}$

$C_2 = \{(1,5),(5,4),(4,6),(6,5),(5,7),(7,2),(2,6),(6,3),(3,4),(4,1)\}$

$C_3 = \{(7,9),(9,8),(8,7)\}$

$C_4 = \{((9,10),(10,8),(8,13),(13,11),(11,9)\}$

$C_5 = \{((9,12),(12,13),(13,14),(14,7),(7,13),(13,9)\}$

$C_6 = \{((14,15),(15,16),(16,14)\}$
Obtain an Euler partition $C = C_1, C_2, \ldots, C_k$ of $G$

Determine the cycle each edge belongs to:
- create array $cycrep$ – it will contain the representative of a cycle each edge belongs to
- apply the BSP/CGM pointer jumping algorithm to obtain array $cycrep$

Representative of the cycle each edge belongs

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>(1,3)(1,5)(1,3)(1,5)(1,3)(1,5)(1,5)(1,5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>(1,5)(1,5)(1,5)(1,5)(7,9)(7,13)(7,9)</td>
</tr>
<tr>
<td>$P_3$</td>
<td>(8,13)(7,13)(7,13)(14,15) (14,15) (14,15)</td>
</tr>
</tbody>
</table>

(1.4) determine the number of cycles of the Euler partition
- each processor computes which of its edges are representatives of cycles: $cyc$
- all processors send $cyc$ to the root processor
- root processor accumulates the $cyc$ of all processors into $ncyc$
- root processor sends $ncyc$ to all processors; if $ncyc=1 \rightarrow$ ET has been found
Step 2: Construct the bipartite graph

(2.1) each processor executes:

for \( i = 0 \) to \( m/p \) do

- \( \text{bipart}(i).u := \text{edge}(i).v \)
- \( \text{bipart}(i).v := \text{cycrep}(i) \)

(2.1) eliminate replicated edges:

<table>
<thead>
<tr>
<th>Processor</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>(1,(1,3)) (1,(1,5)) (2,(1,3)) (2,(1,5)) (3,(1,3)) (3,(1,5)) (4,(1,5))</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>(5,(1,5)) (6,(1,5)) (7,(1,5)) (7,(7,9)) (7,(7,13)) (8,(7,9))</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>(8,(8,13)) (9,(7,9)) (9,(8,13)) (9,(7,13)) (10,(8,13)) (11,(8,13)) (12,(1,5)) (13,(7,13))</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>(13,(8,13)) (14,(7,13)) (14,(14,15)) (15,(14,15)) (14,(14,15))</td>
</tr>
</tbody>
</table>

(2.1) eliminate edges with vertices of degree < 2

<table>
<thead>
<tr>
<th>Processor</th>
<th>Edges</th>
</tr>
</thead>
<tbody>
<tr>
<td>( P_0 )</td>
<td>(1,(1,3)) (1,(1,5)) (2,(1,3)) (2,(1,5)) (3,(1,3)) (3,(1,5))</td>
</tr>
<tr>
<td>( P_1 )</td>
<td>(7,(1,5)) (7,(7,9)) (7,(7,13)) (8,(7,9))</td>
</tr>
<tr>
<td>( P_2 )</td>
<td>(8,(8,13)) (9,(7,9)) (9,(7,13)) (9,(8,13)) (13,(7,13))</td>
</tr>
<tr>
<td>( P_3 )</td>
<td>(13,(8,13)) (14,(7,13)) (14,(14,15))</td>
</tr>
</tbody>
</table>
Bipartite Graph

Construct the Bipartite Graph

Step 3: Construct a Spanning Tree

Obtain a spanning tree of the bipartite graph by the BSP/CGM algorithm Spanning Tree
**Step 3 cont.: Construct a spanning tree**

(3.2) Compute the strut of the spanning tree

Definition 1: A strut $S^* = (V^*, C^*, E^*)$ is a subgraph of a spanning tree $S$ of a bipartite graph $H = (V', C', E')$, such that the degree of each vertex in $V^*$ is greater than or equal to two.

<table>
<thead>
<tr>
<th>$P_0$</th>
<th>$(1,(1,3))$ $(1,(1,5))$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>$(7,(1,5))$ $(7,(7,9))$ $(7,(7,13))$</td>
</tr>
<tr>
<td>$P_2$</td>
<td>$(9,(7,9))$ $(9,(8,13))$</td>
</tr>
<tr>
<td>$P_3$</td>
<td>$(14,(7,13))$ $(14, (14,15))$</td>
</tr>
</tbody>
</table>

(3.3) Compute the number of edges of the strut
Step 4: Perform the stitching

(4.1) If the number of edges to be stitched is equal to $O(m/p)$, processors send their edges to a single processor (root), that performs stitching sequentially.

The stitching consists of exchanging the successors of the edges that belong to the strut:

For each edge of the strut do:

$succ(strut(i)) := succ(strut(i + 1 \ mod \ k))$

where $k$ is the number of edges of the strut that arrive at vertex $v$, for all $v$ that must be stitched.

(4.2) Otherwise, the stitching is performed in a distributed manner.

- Each $P_i$ sorts edges of the strut by $v$ in $V^*$ and computes beginning and end of each sublist (sublist $l$ is formed by the edges of the strut that are incident with the same vertex $v_l$ in $V^*$); $P_i$ sends this info to $P_j$ that stores $v_l$

- Each $P_j$ computes, for each $P_i$ that stores part of sublist $l$, its left and right neighbors: $P_x$ and $P_y$
# Perform the Stitching

<table>
<thead>
<tr>
<th>P₀</th>
<th>vertex: 1 beginning: 0 end: 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>vertex: 7 beginning: 0 end: 2</td>
</tr>
<tr>
<td>P₂</td>
<td>vertex: 9 beginning: 0 end: 1</td>
</tr>
<tr>
<td>P₃</td>
<td>vertex: 14 beginning: 0 end: 1</td>
</tr>
</tbody>
</table>

## Beginning and end of each sublist

<table>
<thead>
<tr>
<th>P₀</th>
<th>vertex: 1 left: 0 right: 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>vertex: 7 left: 1 right: 1</td>
</tr>
<tr>
<td>P₂</td>
<td>vertex: 9 left: 2 right: 2</td>
</tr>
<tr>
<td>P₃</td>
<td>vertex: 14 left: 3 right: 3</td>
</tr>
</tbody>
</table>

## Left and Right neighbors of each sublist

<table>
<thead>
<tr>
<th>P₀</th>
<th>(2,1) (1,3) ⇒ (2,1) (1,5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(4,1) (1,5) ⇒ (4,1) (1,3)</td>
</tr>
<tr>
<td>P₁</td>
<td>(5,7) (7,2) ⇒ (5,7) (7,9)</td>
</tr>
<tr>
<td></td>
<td>(8,7) (&amp;,9) ⇒ (8,7) (7,13)</td>
</tr>
<tr>
<td></td>
<td>(14,7) (7,13) ⇒ (14,13) (7,2)</td>
</tr>
<tr>
<td>P₂</td>
<td>(7,9) (9,8) ⇒ (7,9) (9,10)</td>
</tr>
<tr>
<td></td>
<td>(11,9) (9,10) ⇒ (11,9) (9,8)</td>
</tr>
<tr>
<td>P₃</td>
<td>(13,14) (14,7) ⇒ (13,14) (14,15)</td>
</tr>
<tr>
<td></td>
<td>(16,14) (14,15) ⇒ (16,14) (14,7)</td>
</tr>
</tbody>
</table>
Euler Tour of $G$

**$P_0$**
(2,1) (4,1) (3,2) (7,2) (1,3) (6,3) (3,4) (5,4) (1,5) (1,3) (2,1) (2,6) (3,2) (3,4) (4,1) (4,6)

**$P_1$**
(1,5) (6,5) (2,6) (4,6) (5,7) (8,7) (14,7) (9,8) (5,4) (5,7) (6,3) (6,5) (7,9) (7,13) (7,2) (8,7)

**$P_2$**
(10,8) (7,9) (11,9) (13,9) (9,10) (13,11) (9,12) (7,13) (8,13)(9,10) (9,8) (9,12) (10,8) (11,9) (12,13) (13,9)

**$P_3$**
(8,13) (12,13) (13,14) (16,14) (14,15) (15,16) (13,11) (13,14) (14,15) (14,7) (15,16) (16,14)
**Step 4 cont.: Distributed stitching**

An example where there exist parts of a same sublist distributed among the processors:
Complexity: The BSP/CGM algorithm to compute Euler Tour on an Eulerian graph $G = (V,E)$ uses $O(\log p)$ communication rounds and $O((m+n)/p)$ local computation, with a total of $O(m/p)$ data transmitted.

Results:
Algorithm 1 implemented using C language and MPI library.
The parallel program uses SPMD (Single Program Multiple Data) paradigm.
Concern: to measure running time spent to compute the Euler tour (excluding the time spent on distribution of input data, reporting the output, running the sequential algorithms for sorting, pointer jumping and spanning tree computation).
Experiment carried out in a Beowulf of IC-UNICAMP, with 66 Pentium processors, each one with 256 MB RAM and 256 MB for swapping.
Communication among processors done by a 100 Mbits/s interconnection network.
Time curves for a graph with 512 vertices and 1600 vertices

Starting at about 16 processors, speedup is minimal