The Euler Tour Technique and Parallel Rooted Spanning Tree

Guojing Cong, David A. Bader*
Department of Electrical and Computer Engineering
University of New Mexico, Albuquerque, NM 87131 USA
{cong, dbader}@ece.unm.edu

Abstract

Many parallel algorithms for graph problems start with finding a spanning tree and rooting the tree to define some structural relationship on the vertices which can be used by following problem specific computations. The generic procedure is to find an unrooted spanning tree and then root the spanning tree using the Euler tour technique. With a randomized work-time optimal unrooted spanning tree algorithm and work-time optimal list ranking, finding rooted spanning trees can be done work-time optimally on EREW PRAM w.h.p. Yet the Euler tour technique assumes as "given" a circular adjacency list, it is not without implications though to construct the circular adjacency list for the spanning tree found on the fly by a spanning tree algorithm. In fact our experiments show that this "hidden" step of constructing a circular adjacency list could take as much time as both spanning tree and list ranking combined. In this paper we present new efficient algorithms that find rooted spanning trees without using the Euler tour technique and incur little or no overhead over the underlying spanning tree algorithms.

We also present two new approaches that construct Euler tours efficiently when the circular adjacency list is not given. One is a deterministic PRAM algorithm and the other is a randomized algorithm in the symmetric multiprocessor (SMP) model. The randomized algorithm takes a novel approach for the problems of constructing the Euler tour and rooting a tree. It computes a rooted spanning tree first, then constructs an Euler tour directly for the tree using depth-first traversal. The tour constructed is cache-friendly with adjacent edges in the tour stored in consecutive locations of an array so that prefix-sum (scan) can be used for tree computations instead of the more expensive list-ranking.

Keywords: Spanning Tree, Euler Tour, Parallel Graph Algorithms, Shared Memory, High-Performance Algorithm Engineering.

1 Introduction

Many parallel algorithms for graph problems, e.g., biconnected components, ear decomposition, and planarity testing, at some stage require finding a rooted spanning tree to define some structural relationship on the vertices. The standard approach for finding a rooted spanning tree combines two algorithms: an (unrooted) spanning tree algorithm and rooting a tree using the Euler tour technique, which can be done work-time optimally w.h.p. (see Pettie and Ramachandran [12], and JáJá [8]). In practice, however, there is a gap between the input representations assumed by these two algorithms. For most spanning tree algorithms the input graphs are represented as either adjacency or edge lists, while for the Euler tour technique, a special circular adjacency list is required where additional pointers are added to define an Euler circuit on the Eulerian graph induced by the tree that visits each directed edge exactly once. To convert a plain adjacency list into a circular adjacency list, first the adjacency list of each vertex v is made circular by adding pointers from v_i to $v_{(i+1) \mod d(v)-1}$ where $d(v)$ is the degree of v, and v_i ($0 \leq i \leq d(v)-1$) are the neighbors of v. Then for any edge $<u,v>$ there are cross pointers between the two anti-parallel arcs $<u,v>$ and $<v,u>$. Setting up these cross pointers is the major problem. In the literature, the circular adjacency list is usually assumed as "given" for the Euler tour technique. In the case of the rooted span-

*This work was supported in part by NSF Grants CAREER ACI-00-93039, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654; DARPA Contract NBCH30390004; and DOE Sandia National Laboratories contract 161449.

ning tree problem, however, as the results of the spanning tree algorithm are scattered among the adjacency or edge list, setting up the cross pointers efficiently is no longer trivial, especially when it is sandwiched between two very fast parallel algorithms.

Natural questions then arise: are there better direct techniques to root a tree and what additional work is required to convert an efficient spanning tree algorithm into one that produces a rooted spanning tree. If we consider the rooted spanning tree problem as a single problem instead of two subproblem steps, a new approach reveals itself. Many existing efficient parallel spanning tree algorithms adapt a graft-and-shortcut approach to build a tree in a bottom-up fashion. We observe that grafting defines a natural structural relationship on the vertices. Thus the required information to root a tree is already largely paid for. In Section 2 we present new approaches for finding a rooted spanning tree directly without using the Euler tour technique. The new algorithms incur little or no overhead over the underlying spanning tree algorithms.

The Euler tour technique is one of the basic building blocks for designing parallel algorithms, especially for tree computations (JáJá [8], and Karp and Ramachandran's survey in [15]). For example, pre- and post-order numbering, computing the vertex level, computing the number of descendants, and finding a centroid, can be done work-time optimally on EREW PRAM by applying the Euler tour technique. In Tarjan and Vishkin's biconnected components algorithm [14] that originally introduced the Euler tour technique, the input is an edge list with the cross pointers between twin edges $< u, v >$ and $< v, u >$ established as given. This makes it easy to set up later the cross pointers of the Euler tour defined on a spanning tree, yet setting the circular pointers needs additional work because now a list tail has no information of where the list head is. An edge list with cross pointers is an unusual data structure from which arises the subtle question as to whether we should include the step of setting the cross pointers in the execution time and whether it is appropriate for other spanning tree algorithms. For more natural representations, for example, a plain edge list without cross pointers, Tarjan and Vishkin recommend sorting to set up the cross pointers. After selecting the spanning tree edges, they sort all the arcs $< u, v >$ with $\min(u, v)$ as the primary key, and $\max(u, v)$ as the secondary key. The arcs $< u, v >$ and $< v, u >$ are then next to each other in the resulting list so that the cross pointers can be easily set. We denote the Tarjan-Vishkin approach as Euler-Sort. Our experimental results show that using sorting to set up the cross pointers can take more time than the spanning tree algorithm and list ranking combined. Two new efficient algorithms for the construction of Euler tours without sorting nor given circular adjacency lists are presented in Section 3. The first, a PRAM approach, is based on the graft-and-shortcut spanning tree algorithm, and the second, using a more realistic SMP model, computes a cache-friendly Euler tour where prefix-sum (scan) is used for the tree computation.

Section 4 compares the performance of our new rooted spanning tree algorithms (RST-Graft and RST-Trav) over the spanning tree approach, and our two new Euler tour construction approaches (Euler-Graft and Euler-DFS) versus the Tarjan-Vishkin approach (Euler-Sort).

2 New Parallel Rooted Spanning Tree Algorithms

In this section we present two new rooted spanning tree algorithms without using the Euler tour technique. Both algorithms compute a rooted spanning tree directly without having to go through the intermediate step of finding an unrooted spanning tree. One of the algorithms is based on the "graft and shortcut" spanning tree algorithm (denoted as RST-Graft), and the other is based on a new graph traversal spanning tree algorithm for SMPs (denoted as RST-Trav).

2.1 Rooted Spanning Tree: Graft and Shortcut Approach (RST-Graft)

RST-Graft is based on Shiloach-Vishkin's connected component algorithm (from which a spanning tree algorithm ST-Graft can be derived [1]) that adopts the "graft and shortcut" approach. We observe that ST-Graft can be extended naturally to deal with rooted spanning tree problems. In ST-Graft for a graph $G = (V, E)$ where $|V| = n$ and $|E| = m$ we start with n isolated vertices and m PRAM processors. Each processor P_i ($1 \leq i \leq m$) tries to graft vertex v_i to one of its neighbors u under the constraint that $u < v_i$. Grafting creates $k \geq 1$ connected components in the graph, and each of the k components is then shortcut to a single supervertex. The approach continues to graft and shortcut on the reduced graphs $G' = (V', E')$ with V', the set of supervertices, and E', the set of edges among supervertices, until only one vertex is left.

For a rooted spanning tree algorithm, we need to set up the *parent* relationship on each vertex. Note that grafting defines the *parent* relationship naturally on the

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**

vertices. With RST-Graft we set *parent*(u) to v each time an edge $e = <u, v> \in E$ causes the grafting of the subtree rooted the supervertex of u onto the subtree of v. One issue we face with this approach is that for one vertex its *parent* could be set multiple times by the grafting, hence creating conflicts as shown in Fig. 1.

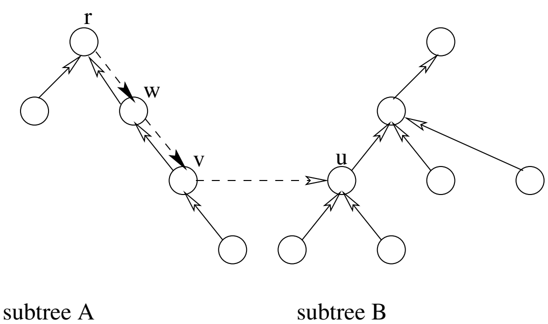

Figure 1. Setting the parent relationship. The *parent* of v was initially set to w, then set to u.

In Fig. 1, after the i^{th} iteration, there are two subtrees A and B, and the white arrows show the current *parent* relationship on the vertices where *parent*(v) is set to w in subtree A. In the $(i+1)^{\text{st}}$ iteration, subtree A is grafted onto subtree B by edge $< u, v >$ and *parent*(v) is set to u as shown by the dashed line with a white arrow. To merge the two subtrees consistently into one rooted spanning tree, we need to reroot subtree A at v and reverse the *parent* relationship for vertices on the path from v to root r of subtree A as shown by the dashed lines with black arrows.

Existing algorithms for rerooting a tree also use the Euler tour technique. Instead we give our new, faster algorithm that uses pointer-jumping and broadcasting. The basic idea is to find all the vertices that are on the path from v to r and reverse their *parent* relationship; that is, if $u = parent(v)$, we now set $parent(u) = v$. Note that simply chasing the *parent* pointer from v to r has $\Theta(H(v))$ complexity where $H(v)$ is the height of v and in the worst case could be $\Theta(n)$ where n is the number of vertices in the tree.

In our algorithm to reroot a tree from r to the new root r', associated with each vertex u in the tree is an array *PR* of size $O(\log n)$. All vertices that ever become u's *parent* in the process of pointer jumping are put into *PR*. Fig. 2 shows an example of *PR* for the vertices after pointer jumping.

Information stored in *PR* is useful when we try to find all the vertices that are on the path from u and root r.

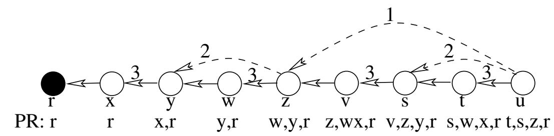

Figure 2. Illustration of *PR* for each vertex after pointer jumping. The number on each edge shows the step during which the message is broadcasted.

We find these vertices in a way that is similar to that of broadcasting over a binary tree. Take Fig. 2 as an example. Here we denote the processor assigned to work on vertex v processor P_v. Except for processor P_r whose vertex is always on some path to r, initially only processor P_u knows that its vertex is on the path. In the first step, P_u checks its *PR* array, finds z on the path, and *broadcasts* a message to processor P_z. In the second step, there are two processors P_u and P_z that are aware that their vertices are on the path, so they both again check their *PR* and broadcast messages to P_y and P_s, respectively. In the step that follows, there will be four processors broadcasting, and so on.

Lemma 1 *Identifying all the vertices on the path from u to root r can be done in $O(\log n)$ time with n processors on EREW PRAM.*

Proof *PR* for each vertex v is created by recording v's parent during pointer-jumping. Pointer-jumping can be done in $O(\log n)$ time with n processors on EREW PRAM. With *PR* available for each vertex v, all the vertices on the path from u to root r can be identified by broadcasting from v in $O(\log n)$ time with n processors on EREW PRAM. □

Lemma 2 *Rerooting a tree can be done in $O(\log n)$ time with n processors on EREW PRAM.*

Proof By Lemma 1, identify the vertices on the path from u to root r. This takes $O(\log n)$ time with n processors on EREW PRAM. Reversing the pointers can be done in $O(1)$ time with n processors by finding a vertex's parent and setting the parent of a vertex's parent to be the vertex itself. □

Alg. 1 in Appendix A is a formal description of our algorithm that reroots a tree from r to u.

For any of the unrooted spanning tree algorithms that adopts the graft and shortcut approach (e.g., spanning tree algorithms based on Shiloach-Vishkin's and

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**

Hirschberg *et al.*'s connected component algorithm [13, 6], see [1] for a survey and comparison), we can extend it to a rooted spanning tree algorithm with Alg. 1. In our RST-Graft algorithm, whenever an edge $< u, v >$ causes the grafting of one subtree A that contains u onto another subtree B that contains v, except when it is the case that u is the root of A, our rooted spanning tree algorithm invokes the tree rerooting algorithm to reroot A at u. Alg. 2 in Appendix A is the formal description of RST-Graft.

Note that Alg. 1 reroots one subtree, while in each iteration of RST-Graft, multiple graftings can happen and multiple subtrees need to be rerooted. We show that rerooting a forest of subtrees can be done within the same complexity for rooting one subtree.

Lemma 3 *A forest of subtrees with a total of n vertices can be rerooted in $O(\log n)$ time with n processors on EREW PRAM.*

Proof For each subtree T_i with size n_i, we can reroot T_i with n_i processors within $O(\log n_i)$ time on EREW by applying the same argument of Lemma 2. With $\sum_i n_i = n$ and $\max_i n_i \leq n$, we can reroot the subtrees within the stated complexity bound. □

Theorem 1 *RST-Graft computes a rooted spanning tree on CRCW PRAM in $O(\log^2 n)$ time with $O(m)$ processors.*

Proof Steps 1, 2, 3 and 4, of Alg. 2 run in $O(1)$ time with m processors; moreover, Step 2 requires arbitrary CRCW PRAM. Step 5 is the pointer-jumping step that runs in $O(\log n)$ time with n processors. Step 6 is the rerooting step that takes $O(\log n)$ time with n processors. During each iteration the number of (super)vertices is reduced at least by half, and since there are $O(\log n)$ iterations, RST-Graft can be performed on an arbitrary CRCW PRAM with $O(\log^2 n)$ time using m processors. □

Note that although RST-Graft has an additional $\log n$ factor compared with the best approach in theory, the $\log n$ factor is determined by both the underlying spanning tree algorithm which we found to be more practical than most other spanning tree algorithms with lower running times, and by the tree rerooting algorithm. Our experimental results show that RST-Graft runs almost as fast as ST-Graft.

2.2 Rooted Spanning Tree: Graph Traversal Approach (RST-Trav)

Previously we designed a parallel spanning tree algorithm that achieves good speedup over the best sequential spanning tree algorithm on SMPs [1]. We observe that this algorithm is also a rooted spanning tree algorithm without any further overhead because it builds a spanning tree by finding the *parent* of each vertex. Here we give a brief description of the algorithm, and refer interested readers to our spanning tree papers for details. First a small rooted subtree T_s with size $O(p)$ is created where p is the number of processors. Then each processor picks a unique leaf l from T_s and starts growing a subtree rooted at l by breadth-first traversal. In the process of breadth-first search, a processor will check a vertex's color, and if it is not colored, color it with a color that is associated with the processor, and set its parent. Under the assumption of sequential memory consistency, the algorithm correctly computes a spanning tree by setting up the parent relation for each vertex in the graph. Without further modification, this algorithm also produces a rooted spanning tree algorithm (RST-Trav), so the accompanying experimental results can also be used for rooted spanning tree. One caveat of RST-Trav is that in some very rare cases, there may be a limit to the parallelism of this approach and a detection scheme allows the algorithm fall back to either Shiloach-Vishkin's or Hirschberg *et al.*'s algorithm. As RST-Graft is a rooted spanning tree algorithm based on Shiloach-Vishkin's algorithm, in case of encountering such graphs, we can fall back to RST-Graft.

3 Efficient Construction of Euler Tours

In this section we show two new approaches for constructing Euler tours without sorting. The first one is in the PRAM framework that finds the twin edge $< v, u >$ of edge $< u, v >$ in the edge list during the spanning tree algorithm, and the second one uses the more realistic SMP model that instead builds a tour from a rooted tree based on the DFS ordering of the edges. We denote the first approach Euler-Graft, the second Euler-DFS.

3.1 Euler Tour Construction: Graft and Shortcut Approach (Euler-Graft)

As with RST-Graft, twin edges can be found with some small overhead during the graft-and-shortcut spanning tree algorithm. The key observation is that for any iteration in the spanning tree algorithm edge $< u, v >$ is inspected and a grafting happens only if v's supervertex is less than u's supervertex and the twin $< x = v, y = u >$ is somewhere else in the edge list. $< x, y >$ would have caused the grafting if the inspection is to compare whether v's supervertex is less than u's superver-

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 IEEE

tex. To identify $< x,y >$ without having to search for it, we re-run the iteration and do the comparison in the other direction. Note that in an iteration we need to find $O(m)$ twin edges. If an edge causes this same grafting as $< u,v >$, we know it is the twin of $< u,v >$. This is true under the condition that in every iteration a supervertex is grafted only by one processor to one other supervertex. To guarantee that this condition holds on CRCW PRAM, basically a tournament is run between the processors that try to graft a certain supervertex v, and only the winner gets to graft v. The graft and shortcut approach runs with $O(\log n)$ time with m processors on arbitrary CRCW [1, 14].

Theorem 2 *With m processors using the graft-and-shortcut spanning tree algorithm, the twin pointers for the Euler tour can be set in $O(\log n)$ time on CRCW PRAM.*

Proof In each iteration of the graft-and-shortcut spanning tree algorithm two inspection rounds are executed. For edge $< u,v >$, the first round checks whether v's supervertex is less than u's supervertex and if $< u,v >$ causes a grafting, associates with u's supervertex the $< u,v >$ location in the edge list. In the second round for edge $< x,y >$ we check whether x's supervertex is less than y's supervertex, and if true, then associate with x's supervertex the location of $< x,y >$'s twin in the edge list. When the algorithm terminates, each edge in the spanning tree finds its twin. The running time is the same as the spanning tree algorithm, which is $O(\log n)$ time with m processors on arbitrary CRCW. □

After the twin pointer is set, we can construct an Euler tour in $O(1)$ time with n processors. This approach can be easily combined with RST-Graft, so with one single run of a graft-and-shortcut spanning tree algorithm, we have spanning tree, rooted spanning tree, and Euler tour, for the tree.

3.2 Euler Tour Construction: DFS Approach (Euler-DFS)

Generally list ranking is needed to perform tree computations with the Euler tour. For an edge $< u,v >$, the next edge $< v,w >$ could be far away from $< u,v >$ in the list with no spatial locality. For any fast implementations on modern computer systems, temporal and spatial locality of the algorithm and the data structures are crucial for good performance. An algorithm with good temporal and spatial locality lends itself to cache-friendly implementations. It is desirable that for an Euler tour the consecutive edges are placed nearby each other in the list. We present a randomized algorithm under the SMP model that constructs an optimal tour in terms of spatial locality, i.e., consecutive edges in the tour are placed into consecutive memory locations in the list. Under the SMP model, there are two parts to an algorithm's complexity, M_E the memory access complexity (number of non-contiguous memory accesses) and T_C the computation complexity [5]. Parameters of the model include the problem size n and the number of processors p. The M_E term recognizes the importance that memory accesses have over an algorithm's performance. Recognizing that p is usually far smaller than n, which is true for most actual problems and architectures, has significant impact on the design of algorithms. Euler-DFS is one such example.

Given a rooted spanning tree T with root r, the basic idea of our algorithm is first to break T into s tree blocks by randomly choosing $s-1$ vertices (called *rep_vertices*). As r is a natural *rep_vertex* the total number of *rep_vertices* is s. The resulting tree blocks are non-overlapping except at the s *rep_vertices*, and the *rep_vertices* form a *rep-tree* (short for representative tree) if we shrink a tree block into a single *rep_vertex*. Fig. 3 illustrates the notion of tree blocks.

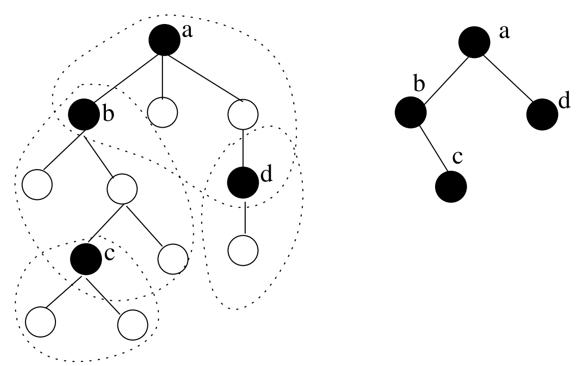

Figure 3. Illustration of tree blocks. On the left, four *rep_vertices* a, b, c, and d, are chosen. Tree blocks are circled by dotted lines. On the right is the *rep-tree*.

Each of these tree blocks is then traversed in DFS order creating s local Euler tours for the s subtrees. We then combine the s local Euler tours into one global tour for which each local tour is broken up at each *rep_vertex* it encounters to incorporate the local tour of the tree block represented by that *rep_vertex*. To do so for each *rep_vertex* we need to compute where its local tour starts in the global tour for T. This is achieved by doing some tree computations on the much smaller *rep-tree*.

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**

We do post-order numbering and DFS numbering on *rep-tree* and record in *local_dfs_num* the DFS numbering of each *rep_vertex* in its parent's tree block. *global_start* is the location in the global tour where *rep_vertex*'s local tour starts, and *g_size* is the size of the subtree (not tree block) rooted at *rep_vertex*. *g_size* can be computed in one post-order traversal of *rep-tree*. Then in the order of DFS numbering, for each *rep_vertex* v with predecessor u and parent w, we set v's *g_size* to be $u.global\_start + v.local\_dfs\_num$ if $u = w$, otherwise set $v.global\_start$ to be $w.global\_start + t + v.local\_dfs\_num$ where t is the sum of the *g_size* of the siblings of v listed before v in the DFS ordering.

The pseudo-code of our algorithm is as follows, in which the input is a rooted tree T with n vertices and root r, and the output is an Euler tour $\mathcal{E}$ with consecutive edges of the tour stored in consecutive memory locations.

- **(1):** For processor P_i $(0 \le i \le p-1)$, if $i\frac{n}{p} \le r \le (i+1)\frac{n}{p} - 1$, choose uniformly and at random $\frac{s}{p} - 1$ non-root vertices as *rep_vertices* and add r to the *rep_vertices*; otherwise, choose uniformly and at random $\frac{s}{p}$ vertices as *rep_vertices*. The result is that T is broken into s tree blocks $B_1, \cdots, B_s$, with each processor owning the *rep_vertices* of $\frac{s}{p}$ blocks.

- **(2):** Processor P_i traverses each of its $\frac{s}{p}$ tree blocks rooted at the *rep_vertices* in DFS order, creating $\frac{s}{p}$ local Euler tours of the blocks. With each edge e of the tour is associated a *position* that now records the location of e in the local tour. With each *rep_vertex* is associated a *local_dfs_num* that records the DFS numbering of the *rep_vertex* in its parent's tree block.

- **(3):** Processor P_0 computes post-order numbering and DFS numbering of the *rep-tree* formed by the s *rep_vertices*. In one post-order traversal, the subtree size *g_size* rooted at each *rep_vertex* can be computed. For root r, $r.global\_start = 0$. In the order of DFS numbering, if a *rep_vertex* v has a predecessor u that is also its parent w $(u = w)$, which means from r to v the global tour is the pieces of local tours that have been seen during the DFS traversal, then set $v.global\_start = w.global\_start + v.local\_dfs\_num$; otherwise extra spaces are needed to accommodate other local tours. One way to look at this is w's *global_start* is now available, and we know the DFS numbering of v in w's tree block, so there must have been other *rep_vertices* $\mathcal{S}$ visited in DFS order before v (or otherwise $u = v$). Hence, extra spaces are needed to accommodate the tours of the subtrees (not blocks) rooted at the vertices in $\mathcal{S}$, and they are v's siblings. Let t be the sum of the *g_size* of the siblings of v listed before v in the DFS ordering in w's tree block. $v.global\_start = w.global\_start + t + v.local\_dfs\_num.$

- **(4):** For each of the $\frac{s}{p}$ tours defined by the *rep_vertices* on processor P_i, for each edge e in the ordering of the appearance in the local tour, if $e = \langle u, v \rangle$ with u being a *rep_vertex*, add $\delta = u.global\_start$ to *position* of e and all edges following e until the appearance of another *rep_vertex*; if $e = \langle u, v \rangle$ with v being a *rep_vertex*, increase δ by $v.g\_size$.

- **(5):** For each of the $\frac{s}{p}$ tours defined by the *rep_vertices* on processor P_i, copy each edge e in the tour into location $e.position$ of tour $\mathcal{E}$.

Let $|B|$ be the size of block B, that is, the number of vertices in the tree block. We show that the largest tree block B_i has size less than $\frac{cn}{s}$ (where c is some small constant greater than one) with probability at least $1 - e^{-c}$.

Lemma 4

$$P\left(|B_i| \ge \frac{cn}{s}\right) \le e^{-c}.$$

Proof Suppose T_i is the smallest subtree of T that contains B_i. As each vertex is equally likely to be a *rep_vertex*, the probability that a *rep_vertex* is outside of T_i is

$$\frac{n - \frac{cn}{s}}{n} = \frac{s-c}{s},$$

while the probability that a *rep_vertex* is inside T_i is $\frac{c}{s}$. Let

$\mathcal{A}$ be the event that $|B_i| \ge \frac{cn}{s}$,
$\mathcal{B}$ be the event that no *rep_vertex* splits T_i ($\mathcal{A} \subseteq \mathcal{B}$),
Θ_k be the event that k *rep_vertices* fall in T_i, and
Λ_k be the event that k *rep_vertices* fall outside T_i.

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 IEEE

$$
\begin{aligned}
P\left(|B_i| \geq \frac{cn}{s}\right) &= P(\mathcal{A}) \leq P(\mathcal{B}) \\
&\leq \sum_{k=1}^{s} P(\Theta_k) P(\Lambda_{s-k}) \\
&= \sum_{k=1}^{s} \left(\frac{s-c}{s}\right)^{s-k} \left(\frac{c}{s}\right)^{k} \\
&= \left(\frac{s-c}{s}\right)^{s} \sum_{k=1}^{s} \left(\frac{c}{s-c}\right)^{k} \\
&\leq \left(\frac{s-c}{s}\right)^{s} \\
&\leq e^{-c}
\end{aligned}
$$

□

The expected size of a tree block is $\frac{n+s-1}{s} \approx \frac{n}{s}$ when n is large. By Lemma 4 the probability that the largest number of vertices visited by a particular processor deviates from $\frac{n}{p}$ by a constant factor of α is bounded by $e^{(-\alpha s)/p}$, which can be bounded by $n^{-\lambda}$ for some $\lambda > 0$ if $\frac{s}{p} \geq \ln n$. Hence with high probability the work is fairly balanced among the processors, and the expected running time of the algorithm is $O(n/p + \ln n)$ with p processors when $n \gg p$ and $s = p \ln n$.

4 Experimental Results

This section summarizes the experimental results of our implementation. We compare the performance of ST-Graft and RST-Graft on a variety of input graphs. As for RST-Trav, [1] give a comprehensive experimental study. We test our shared-memory implementation on the Sun Enterprise 4500, a uniform memory access (UMA) shared memory parallel machine with 14 UltraSPARC II processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 MBytes of external (L2) cache. The clock speed of each processor is 400 MHz.

4.1 Experimental Data

Next we describe the collection of graph generators that we use to compare the performance of the parallel rooted spanning tree graph algorithms. Our generators include several employed in previous experimental studies of parallel graph algorithms for related problems. For instance, we include the mesh topologies used in the connected component studies of [4, 9, 7, 3], the random graphs used by [4, 2, 7, 3], the geometric graphs used by [2], and the “tertiary” geometric graph **AD3** used by [4, 7, 9, 3].

- **Meshes** Computational science applications for physics-based simulations and computer vision commonly use mesh-based graphs. In the **2D Mesh**, the vertices of the graph are placed on a 2D mesh, with each vertex connected to its four neighbors.
- **Random Graph** We create a random graph of n vertices and m edges by randomly adding m unique edges to the vertex set. Several software packages generate random graphs this way, including LEDA [10].
- **Geometric Graphs and AD3** In these k-regular graphs, n points are chosen uniformly and at random in a unit square in the Cartesian plane, and each vertex is connected to its k nearest neighbors. Moret and Shapiro [11] use these in their empirical study of sequential MST algorithms. **AD3** is a geometric graph with $k = 3$.

4.2 Performance Results and Analysis

Fig. 4 contains the performance comparison between the rooted spanning tree algorithm RST-Graft, Euler tour construction algorithm Euler-Graft and the spanning tree algorithm ST-Graft. These algorithms are all based on the graft-and-shortcut approach. In the figure the ratios of the running time of RST-Graft/ST-Graft and Euler-Graft/ST-Graft are given. For most input graphs, the overhead of the RST-Graft over ST-Graft is within 25%, and for each input graph, there are cases that RST-Graft actually runs faster than ST-Graft which is caused by the different number of iterations due to the races among the processors. In each iteration of ST-Graft, there are two runs of grafting, the first one is a competition run and the second one actually performs the grafting. In Euler-Graft, a third run is needed to find the twin edges for the tree edges, so we expect to see roughly 50% overhead of Euler-Graft/ST-Graft. This is true for the input graphs shown in Fig. 4. 2D Mesh is an anomaly where Euler-Graft actually runs faster than ST-Graft, because the optimization (compacting edge list) for ST-Graft does not work efficiently for this special input graph.

In Fig. 5 we plot the running times of the two approaches for computing the rooted spanning tree and Euler tour. Euler-DFS is our approach using ST-Trav from [1], and Euler-Sort is based on Tarjan and Vishkin’s approach although we replace the spanning tree algorithm with our much faster algorithm ST-Trav. Generally Euler-DFS is 3-5 times faster than Euler-Sort. Note that the comparison result would be more dramatic if we

Proceedings of the 2004 International Conference on Parallel Processing (ICPP’04)
0190-3918/04 $20.00 © 2004 **IEEE**

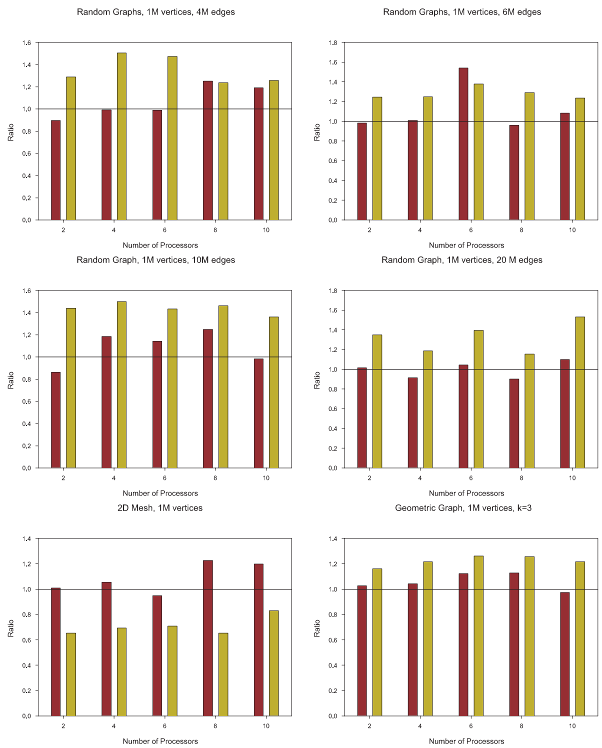

Figure 4. The performance ratio of the running times of RST-Graft and Euler-Graft over the spanning tree algorithm ST-Graft. In each plot, the left- and right-hand bars correspond to RST-Graft and Euler-Graft, respectively, as compared with ST-Graft.

compare only the two approaches for rooted spanning tree, our rooted spanning tree algorithm could be an order of magnitude faster than the traditional approach of spanning tree plus Euler tour. In practice this comparison would be fair if after finding a rooted spanning tree the algorithm no longer does any tree computation. If it does, as in the case of Tarjan and Vishkin's biconnected component algorithm, we compare two algorithms that achieve the same functionality, that is, they both compute the rooted spanning tree and Euler tour. In addition to running faster, Euler-DFS tends to be more flexible (only computes Euler tour when needed), simpler (no sorting is involved) and uses less memory (no need to maintain the twin information). One other advantage of Euler-DFS that does not affect the performance but is a useful property is that it takes the straightforward adjacency list representation of the input graphs. While Euler-Sort needs both adjacency list (required by the fast spanning tree algorithm otherwise it is even slower) and

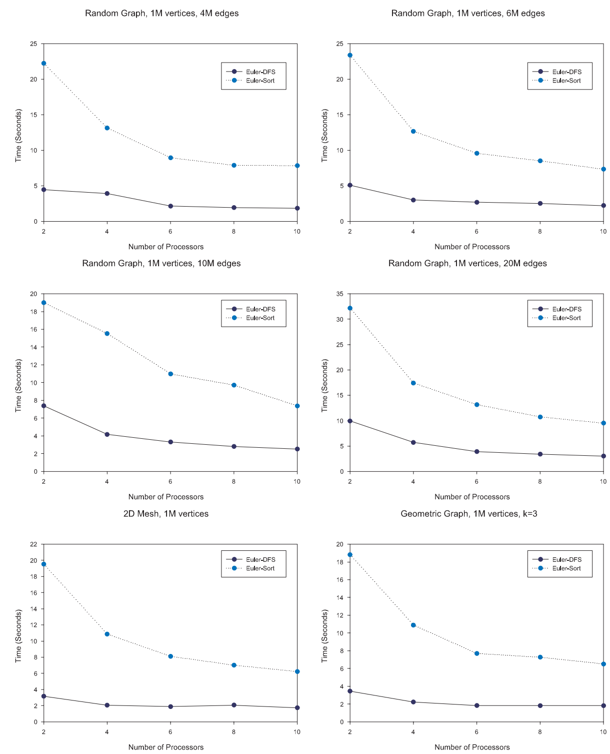

Figure 5. Comparison of Euler-DFS and Euler-Sort running times for various input graphs.

edge list representation, which is clumsy and doubles the memory usage. With Euler-DFS, as the number of processors increases, it is observed that parallel memory writes (e.g., copying tour elements into appropriate locations) start to dominate the execution time. We expect better scalability results of Euler-DFS on platforms with more scalable memory bandwidth.

5 Conclusions

We presented new algorithms for rooted spanning trees and rerooting a tree without having to compute the Euler tour. These algorithms have little or no overhead over the underlying spanning tree algorithms, and the technique will greatly reduce the running time and programming complexity of an algorithm if no further tree computations are required. Two new approaches of computing an Euler tour are also discussed when the pointers between twin edges are not given. For all the input graphs we tested, Euler-DFS runs much faster than the sorting approach. As Euler tour is fundamental in tree computations, our results will have impact on the

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**

implementation of higher-level algorithms, for example, tree computations, biconnected components, lowest common ancestors, upward accumulation, etc. Our results are also examples that algorithm engineering techniques pay off in parallel computing.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[2] S. Chung and A. Condon. Parallel implementation of Borůvka's minimum spanning tree algorithm. In *Proc. 10th Int'l Parallel Processing Symp. (IPPS'96)*, pages 302–315, April 1996.

[3] S. Goddard, S. Kumar, and J.F. Prins. Connected components algorithms for mesh-connected parallel computers. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 43–58. American Mathematical Society, 1997.

[4] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proc. 6th Ann. Symp. Parallel Algorithms and Architectures (SPAA-94)*, pages 16–25, Cape May, NJ, June 1994.

[5] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[6] D. S. Hirschberg, A. K. Chandra, and D. V. Sarwate. Computing connected components on parallel computers. *Commununications of the ACM*, 22(8):461–464, 1979.

[7] T.-S. Hsu, V. Ramachandran, and N. Dean. Parallel implementation of algorithms for finding connected components in graphs. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 23–41. American Mathematical Society, 1997.

[8] J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley Publishing Company, New York, 1992.

[9] A. Krishnamurthy, S. S. Lumetta, D. E. Culler, and K. Yelick. Connected components on distributed memory machines. In S. N. Bhatt, editor, *Parallel Algorithms: 3rd DIMACS Implementation Challenge October 17-19, 1994*, volume 30 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 1–21. American Mathematical Society, 1997.

[10] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[11] B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimal spanning tree. In *DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science: Computational Support for Discrete Mathematics* ***15***, pages 99–117. American Mathematical Society, 1994.

[12] S. Pettie and V. Ramachandran. A randomized time-work optimal parallel algorithm for finding a minimum spanning forest. *SIAM J. Comput.*, 31(6):1879–1895, 2002.

[13] Y. Shiloach and U. Vishkin. An $O(\log n)$ parallel connectivity algorithm. *J. Algs.*, 3(1):57–67, 1982.

[14] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM J. Computing*, 14(4):862–874, 1985.

[15] J. van Leeuwen, editor. *Handbook of Theoretical Computer Science*. The MIT Press/Elsevier, New York, 1990.

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**

A Algorithms

Input: 1. A rooted tree T with n vertices, root r, and new root u
2. Array PR_v for each vertex v

Output: Rooted tree T' with root at u

begin
  if $i = u$ **then** On-Path[i] $\leftarrow$1 **else** On-Path[i] $\leftarrow$ 0 ;
  $j \leftarrow size(PR_i) - 1$;
  for $h \leftarrow 1$ *to* $\lceil \log n \rceil$ **do**
    if *On-Path[i]=1 and* $j \geq 1$ **then**
      On-Path[$PR_i[j]$] $\leftarrow 1$;
      $j \leftarrow j - 1$;
  if *On-Path[i]=1* **then**
    $parent[parent[i]] \leftarrow i$
end

Algorithm 1: Algorithm for rerooting a tree on processor P_i, for $(0 \leq i \leq n-1)$

Input: Undirected graph $G = (V, E)$ with n vertices, m edges

Output: A rooted spanning tree T of G

begin
  while *Number of Connected Component* > 1
  do
    1. **for** $i \leftarrow 1$ *to n in parallel* **do** $D[i] \leftarrow i$;
    2. **for** $i \leftarrow 1$ *to n in parallel* **do**
      for *each neighbor j of i in parallel* **do**
        if $D[j] < D[i]$ **then**
          $Winner[D[i]] \leftarrow K$
    3. **for** $i \leftarrow 1$ *to n in parallel* **do**
      for *each neighbor j of i in parallel* **do**
        if $D[j] < D[i]$ *AND* $Winner[D[i]]$ $=K$ **then**
          $D[D[i]] \leftarrow D[j]$;
          $parent[i] \leftarrow j$;
          Label i as the new root of the old subtree;
    4. j=0;
    5. **for** $i \leftarrow 1$ *to n in parallel* **do**
      if $D[i] \neq D[D[i]]$ **then**
        $PR[j] \leftarrow D[i]$;
        $j \leftarrow j + 1$;
    6. **for** *each i that is labeled as the new root of a subtree* **do**
      call Alg. 1 to reroot the subtree
end

Algorithm 2: Algorithm for finding rooted spanning tree on processor K

Proceedings of the 2004 International Conference on Parallel Processing (ICPP'04)
0190-3918/04 $20.00 © 2004 **IEEE**