An Experimental Study of Parallel Biconnected Components Algorithms on Symmetric Multiprocessors (SMPs)

Guojing Cong
IBM T.J. Watson Research Center
Yorktown Heights, NY 10598 USA

David A. Bader *
College of Computing
Georgia Institute of Technology

February 25, 2006

Abstract

We present an experimental study of parallel biconnected components algorithms employing several fundamental parallel primitives, e.g., prefix sum, list ranking, sorting, connectivity, spanning tree, and tree computations. Previous experimental studies of these primitives demonstrate reasonable parallel speedups. However, when these algorithms are used as subroutines to solve higher-level problems, there are two factors that hinder fast parallel implementations. One is parallel overhead, i.e., the large constant factors hidden in the asymptotic bounds; the other is the discrepancy among the data structures used in the primitives that brings non-negligible conversion cost. We present various optimization techniques and a new parallel algorithm that significantly improve the performance of finding biconnected components of a graph on symmetric multiprocessors (SMPs). Finding biconnected components has application in fault-tolerant network design, and is also used in graph planarity testing. Our parallel implementation achieves speedups up to 4 using 12 processors on a Sun E4500 for large, sparse graphs, and the source code is freely-available at our web site `http://www.ece.unm.edu/~dbader`.

*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, DBI-0420513, ITR EF/BIO 03-31654 and DBI-04-20513; and DARPA Contract NBCH30390004.

1 Introduction

A connected graph is said to be *separable* if there exists a vertex v such that removal of v results in two or more connected components of the graph. Given a connected, undirected graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, the biconnected components problem finds the maximal induced subgraphs of G that are not *separable*. Tarjan [19] presents an optimal $O(n + m)$ algorithm that finds the biconnected components of a graph based on depth-first search (DFS). Eckstein [7] gave the first parallel algorithm that takes $O\left(d \log^2 n\right)$ time with $O((n + m)/d)$ processors on CREW PRAM, where d is the diameter of the graph. Savage and JáJá [17] designed two parallel algorithms on CREW PRAM. The first one takes $O\left(\log^2 n\right)$ time with $O(n^3/\log n)$ processors. The second one is suitable for sparse graphs, and requires $O\left(\log^2 n \log k\right)$ time and $O(mn + n^2 \log n)$ processors where k is the number of biconnected components in the graph. Tsin and Chin [21] developed an algorithm on CREW PRAM that takes $O\left(\log^2 n\right)$ time with $O\left(n^2/\log^2 n\right)$ processors and is optimal for dense graphs. Tarjan and Vishkin [20] present an $O(\log n)$ time algorithm on CRCW PRAM that uses $O(n + m)$ processors. The fundamental Euler-tour technique is also introduced in [20]. Liang *et al.* [14] studied the biconnected components problems for graphs with special properties, e.g., interval graphs, circular-arc graphs and permutation graphs and achieved better complexity bounds. There are also other biconnected components related studies, e.g., finding the smallest augmentation to make a graph biconnected [11], and finding the smallest biconnected spanning subgraph (an NP-hard problem) [13, 5].

Woo and Sahni [22] presented an experimental study of computing biconnected components on a hypercube for Tarjan and Vishkin's algorithm and Read's algorithm [16]. Their

test cases are graphs that retain 70 and 90 percent edges of the complete graphs, and they achieved parallel efficiencies up to 0.7 for these dense inputs. The implementation uses an adjacency matrix as the input representation, and the size of the input graphs is limited to less than 2,000 vertices.

In this paper we present an experimental study of adapting the Tarjan-Vishkin biconnected components algorithm to run on symmetric multiprocessors (SMPs) solving sparse, irregular graph instances. The algorithm is representative of many parallel algorithms that take drastically different approaches than the sequential algorithm to solve certain problems and employ basic parallel primitives such as prefix sum, pointer jumping, list ranking, sorting, connected components, spanning tree, Euler-tour construction and tree computations, as building blocks. Previous studies demonstrated reasonable parallel speedups for these parallel primitives on SMPs [9, 2, 3, 6, 4]. It is not clear whether an implementation using these techniques achieves good speedup compared with the best sequential implementation because of the cost of parallel overheads. Here we focus on algorithmic overhead instead of communication and synchronization overhead. For example, Tarjan's sequential biconnected components algorithm [19] uses DFS with an auxiliary stack, while the Tarjan-Vishkin parallel algorithm (denoted as TV in this paper) employs all the parallel techniques mentioned earlier. Another factor that makes it hard to achieve good parallel speedups is the discrepancies among the input representations assumed by different primitives. TV finds a spanning tree, roots the tree, and performs various tree computations. Algorithms for finding spanning trees take edge list or adjacency list data structures as input representations, while rooting a tree and tree computations assume an Eulerian circuit for the tree that is derived from a

circular adjacency list representation. Converting representations is not trivial and incurs a real cost in implementations. In our studies, direct implementation of TV on SMPs does not outperform the sequential implementation even at 12 processors. In our optimized adaptation of TV onto SMPs, we follow the major steps of TV, yet we use different approaches for several of the steps. For example, we use a different spanning tree algorithm and new approaches to root the tree, construct the Euler-tour and perform the tree computations. With our new algorithm design and engineering techniques, our optimized adaptation of TV achieves speedups up to 2.5 when employing 12 processors.

We also present a new parallel algorithm that eliminates edges that are not essential in computing the biconnected components. For any input graph, edges are first eliminated before the computation of biconnected components is done so that at most $\min(m, 2n)$ edges are considered. Although applying this filtering algorithm does not improve the asymptotic complexity, in practice, the performance of the biconnected components algorithm can be significantly improved. In fact we achieve speedups up to 4 with 12 processors using the filtering technique. This is remarkable, given that the sequential algorithm runs in linear time with a very small hidden constant in the asymptotic complexity.

The organization of this paper is as follows. Section 2 introduces TV; section 3 discusses its implementation and optimization for SMPs; section 4 presents our new edge-filtering algorithm; section 5 gives analysis and performance results; and in section 6 we conclude.

2 The Tarjan-Vishkin Algorithm

First we give a brief review of the Tarjan-Vishkin biconnected components algorithm. For an undirected, connected graph $G = (V, E)$ and a spanning tree T of G, each nontree edge introduces a simple cycle that itself is biconnected. If two cycles C_1 and C_2 share an edge, then $C_1 \cup C_2$ are biconnected. Let R_c be the relation that two cycles share an edge, then the transitive closure of R_c (denoted as R_c^*) partitions the graph into equivalence classes of biconnected components. If we are able to compute R_c, we can find all the biconnected components of graph G.

The size of R_c is too large ($O(n^2)$ even for sparse graphs where $m = O(n)$) to be usable in fast parallel algorithms. Tarjan and Vishkin defined a smaller relation R_c' with $|R_c'| = O(m)$ and proved that the transitive closure of R_c' is the same as that of R_c [20, 12]. For any pair (e, g) of edges, $(e, g) \in R'$ (or simply denoted as $eR_c'g$) if and only if one of the following three conditions holds (denote the parent of a vertex u as $p(u)$, and the root of T as r):

1. $e = (u, p(u))$ and $g = (u, v)$ in $G - T$, and $v < u$ in preorder numbering
2. $e = (u, p(u))$ and $g = (v, p(v))$, and (u, v) in $G - T$ such that u and v are not related (having no ancestral relationships)
3. $e = (u, p(u))$, $v \neq r$, and $g = (v, p(v))$, and some nontree edge of G joins a descendant of u to a nondescendant of v

Once R_c' is computed, TV builds an auxiliary graph $G' = (V', E')$ where V' is the set E of edges of G, and $(e, g) \in E'$ if $eR_c'g$. The connected components of G' correspond to the equivalence classes of $R_c'^*$ and identify the biconnected components of G.

TV has six steps:

1. *Spanning-tree* computes a spanning tree T for the input graph G. A spanning tree algorithm derived from the Shiloach-Vishkin's connected components algorithm [18] is used.

2. *Euler-tour* constructs an Eulerian circuit for T.

3. *Root-tree* roots T at an arbitrary vertex by applying the Euler-tour technique on the circuit obtained in the previous step.

4. *Low-high* computes two values low(v) and high(v) for each vertex v. The value low(v) denotes the smallest vertex (in preorder numbering) that is either a descendant of v or adjacent to a descendent of v by a nontree edge. Similarly, high(v) denotes the largest vertex (in preorder numbering) that is either a descendant of v or adjacent to a descendent of v by a nontree edge.

5. *Label-edge* tests the appropriate conditions of R'_c and builds the auxiliary graph G' using the low, high values.

6. *Connected-components* finds the connected components of G' with the Shiloach-Vishkin connected components algorithm.

TV takes an edge list as input. The parallel implementations of the six steps within the complexity bound of $O(\log n)$ time and $O(m)$ processors on CRCW PRAM are straightforward except for the *Label-edge* step. Tarjan and Vishkin claim that the *Label-edge* step takes constant time with $O(m)$ processors because the conditions for R'_c can be tested within

these bounds. Note that if two edges e and g satisfy one of the conditions for R'_c, mapping $(e, g) \in R'_c$ into an edge in $G' = (V', E')$ is not straightforward because no information is available about which vertices e and g are mapped to in V'. Take condition 1 for example. For each nontree edge $g_1 = (u, v) \in E$, if $u < v$ and let $g_2 = (u, p(u))$, (g_1, g_2) maps to an edge in E'. If we map each edge $e \in E$ into a vertex $v' \in V'$ whose number is the location of e in the edge list, we need to search for the location of g_2 in the edge list.

Input: L: an edge list representation for graph $G = (V, E)$ where $|V| = n$ and $|E| = m$
Preorder: preorder numbering for the vertices
Output: G': an edge list representation of the auxiliary graph
begin
  for $0 \leq i \leq m-1$ *parallel* **do**
    if *L[i].is_tree_edge=true* **then** $N[i] \leftarrow 1$;
    else $N[i] \leftarrow 0$;
  prefix-sum(N,m);
  for $0 \leq i \leq m-1$ *parallel* **do**
    u=$L[i].v1$; v=$L[i].v2$;
    if *L[i].is_tree_edge=true* **then**
      if *Preorder[v]<Preorder[u]* **then** $L'[i] \leftarrow (u, N[i]+n)$;
      if *u and v not related* **then** $L'[m+i] \leftarrow (u,v)$;
    else
      if *u* $\neq$ *root and v* $\neq$ *root* **then** $L'[2m+i] \leftarrow (u,v)$;
  compact L' into G' using prefix-sum;
end

Algorithm 1: building the auxiliary graph.

Here we present an algorithm for this missing step in TV that builds the auxiliary graph in $O(\log m)$ time with $O(m)$ processors, which does not violate the claimed overall complexity bounds of TV. The basic idea of the algorithm is as follows. Assume, w.l.o.g., $V = [1, n]$ (In this paper we use $[a, b]$ denote the integer interval between a and b). $V' = [1, m]$. We map each tree edge $(u, p(u)) \in E$ to vertex $u \in V'$. For each nontree edge e, we assign a distinct integer n_e between $[0, m-n]$, and map e to vertex $n_e + n \in V'$. Assigning numbers to nontree

edges can be done by a prefix sum. The formal description of the algorithm is shown in Alg. 1.

We prove Alg. 1 builds an auxiliary graph within the complexity bound of TV.

Theorem 1 *Alg. 1 builds an auxiliary graph $G' = (V', E')$ in $O(\log m)$ time with $O(m)$ processors and $O(m)$ space on EREW PRAM.*

Proof: According to the mapping scheme, $V' = [1, m+n]$. Each tree edge $L[i] = (u, p(u))$ is uniquely mapped to $u \in V'$. For each nontree edge $L[j]$, a unique number $N[j] \in [1, m]$ is assigned. Nontree edge $L[j]$ is mapped to $N[j] + n \in V'$ so that it is not mapped to a vertex number assigned to a tree edge and no two nontree edges share the same vertex number. It is easy to verify that this is a one-to-one mapping from E to V' and can be done in $\mathrm{O}(\log m)$ time with $\mathrm{O}(m)$ processors. As for E', testing the conditions, i.e., finding all the edges in E', can be done in constant time with $\mathrm{O}(m)$ processors.

A complication arises in the determination of where to store the edge information each time we add a new edge e' (image of (e, g) where $e, g \in E$) to E'. A straightforward approach uses an $(n+m) \times (n+m)$ matrix so that each edge of E' maps to a unique location. A better, space-efficient approach is as follows. If we inspect the conditions for R'_c closely, we see that for each condition we add at most m edges to the edge list. L' is a temporary structure that has $3m$ locations. Locations $[0, m-1]$, $[m, 2m-1]$, and $[2m, 3m-1]$, are allocated for condition 1, 2 and 3, respectively. After all the edges are discovered, L' is compacted into G' using prefix sums. Prefix sums dominate the running time of Alg. 1, and no concurrent reads or writes are required. So Alg. 1 builds G' (the auxiliary graph) in $\mathrm{O}(\log m)$ time with $\mathrm{O}(m)$ processors and $\mathrm{O}(m)$ space on EREW PRAM. □

3 Implementation and optimization

In this section we show our adaptation of TV on SMPs (*TV-SMP*) and an optimized version of the Tarjan-Vishkin algorithm (*TV-opt*). *TV-SMP* emulates TV on SMPs, and serves as a baseline implementation for comparison with the optimized version and our new algorithm. *TV-opt* optimizes TV to run on SMPs by reorganizing some of the steps of TV and substituting several steps with more efficient algorithms.

3.1 *TV-SMP*

TV-SMP emulates TV in a coarse-grained fashion by scaling down the parallelism of TV to the number of processors available from an SMP. The emulation of each step is straightforward except for the *Euler-tour* step. In the literature the Euler-tour technique usually assumes a circular adjacency list as input where there are cross pointers between the two anti-parallel arcs (u, v) and (v, u) of an edge $e = (u, v)$ in the edge list. For the tree edges found in the *Spanning-tree* step, such a circular adjacency list has to be constructed on the fly. The major task is to find for an arc (u, v) the location of its anti-parallel mate, (v, u). After selecting the spanning tree edges, we 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 use the efficient parallel sample sorting routine designed by Helman and JáJá [8]. Our experimental study shows that the parallel overheads of *TV-SMP* are too high for the implementation to achieve parallel speedup with a moderate number of processors. Detailed performance results are given in section 5.

3.2 *TV-opt*

With *TV-opt* we optimize *TV* to run on SMPs by using algorithm engineering techniques to reduce the parallel overhead. Two major optimizations are considered. First we reduce the number of parallel primitives and subroutines used in our implementation by rearranging and merging some of the steps. Second we substitute the algorithms for certain steps with more efficient and cache-friendly versions.

We merge the *Spanning-tree* and *Root-tree* steps because a rooted spanning tree algorithm can usually be derived from the spanning tree algorithm with very little overhead. In our previous studies [6], we propose parallel algorithms that compute a rooted spanning tree for an input graph directly without invoking the standard Euler-tour technique. With any spanning tree algorithm that adapts the "graft and shortcut" approach (e.g., the Shiloach-Vishkin algorithm (SV) [18, 1], and Hirschberg *et al.*'s algorithm (HCS) [10]), we observe that grafting defines the *parent* relationship naturally on the vertices (extra care needs to be taken to resolve the conflicts when a vertex's *parent* is set multiple times by grafting). Better still is our work-stealing graph-traversal spanning tree algorithm that computes a spanning tree (also a rooted spanning tree) by setting the *parent* for each vertex. Our algorithm achieves superior speedup over the best sequential algorithms (BFS or DFS, which also compute a rooted spanning tree) compared with other spanning tree algorithms (e.g., SV and HCS). We refer interested readers to [6, 2] for details.

With a rooted spanning tree, we construct a cache-friendly Euler-tour for the tree (Euler-tour is needed for the preorder numbering of vertices). 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 tour with no spatial locality, which hinders cache performance. It is desirable that for an Euler-tour the consecutive edges are placed nearby each other in the list. As an Euler-tour for a tree is essentially a DFS traversal of the tree, we construct the tour based on DFS traversal. A formal description of the algorithm and complexity bound proof are given in [6]. With high probability, the algorithm runs in $O\left(\frac{n}{p}\right)$ time. The algorithm produces an Euler-tour where prefix sum can be used for tree computations instead of the more expensive list ranking.

The remaining steps of *TV-opt* are the same as those of *TV-SMP*. We compare the performances of *TV-opt* and *TV-SMP*, and demonstrate the effect of the optimizations in section 5.

4 A New Algorithm and Further Improvement (*TV-filter*)

The motivation to further improve TV comes from the following observation for many graphs: not all nontree edges are necessary for maintaining the biconnectivity of the biconnected components. We say an edge e is *non-essential* for biconnectivity if removing e does not change the biconnectivity of the component to which it belongs. Filtering out *non-essential* edges when computing biconnected components (of course we will place these edges back in later) may produce performance advantages. Recall that the goal of TV is to find R'^{*}_{c}. Of the three conditions for R'_c, it is trivial to check for condition 1 which is for a tree edge and a non-tree edge. Conditions 2 and 3, however, are for two tree edges and checking involves the computation of *high* and *low* values. To compute *high* and *low*, we

need to inspect every nontree edge of the graph, which is very time consuming when the graph is not extremely sparse. The fewer edges the graph has, the faster the *Low-high* step. Also when we build the auxiliary graph, the fewer edges in the original graph means the smaller the auxiliary graph and the faster the *Label-edge* and *Connected-components* steps.

Take Fig. 1 for example. On the left in Fig. 1 is a biconnected graph G_1. After we remove nontree edges e_1 and e_2, we get a graph G_2 shown on the right in Fig. 1, which is still biconnected. G_1 has a R_c' relation of size 11 (4, 4, and 3 for conditions 1, 2, and 3, respectively), while graph G_2 has a R_c' relation of size 7 (2, 2, and 3 for conditions 1, 2, and 3, respectively). So the auxiliary graph of G_1 has 10 vertices and 11 edges, while the auxiliary graph for G_2 has only 8 vertices and 7 edges. When there are many *non-essential* edges, filtering can greatly speedup the computation.

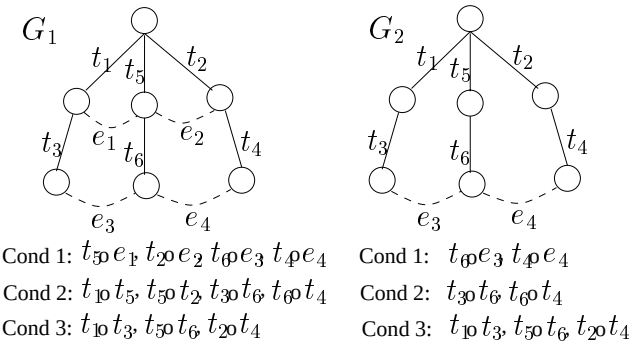

Figure 1: Two graphs G_1 and G_2. The solid edges are tree edges and the dashed edges are nontree edges. G_2 is derived from G_1 by removing *non-essential* nontree edges e_1 and e_2. Below the graphs are the corresponding R_c' relations defined by the three conditions.

Now the questions are how many edges can be filtered out and how to identify the *non-essential* edges for a graph $G = (V, E)$ with a spanning tree T. We postpone the discussion of the first question until later in this section because it is dependent on how filtering is done. First we present an algorithm for identifying *non-essential* edges. The basic idea is to

compute a spanning forest F for $G - T$. We note that if T is a breadth-first search (BFS) tree, then the nontree edges of G that are not in F can be filtered out.

Assuming T is a BFS tree, next we prove several lemmas.

Lemma 1 *For any edge $e = (u, v)$ in F, there is no ancestral relationship between u and v in T.*

Proof: Clearly u and v cannot be the parent of each other as e is not in T. Suppose w.l.o.g. that u is an ancestor of v, and w is the parent of v ($w \neq u$), considering the fact that T is a BFS tree, v is at most one level away from u and w is at least one level away from u. So w cannot be v's parent, and we get a contradiction. □

Lemma 2 *Each connected component of F is in some biconnected component of graph G.*

Proof: Let C be a connected component of F. Note that C itself is also a tree. Each edge in C is a nontree edge to T, and is in a simple cycle, hence some biconnected component, of G. We show by induction that the simple cycles determined by the edges of C form one biconnected component.

Starting with an empty set of edges, we consider the process of growing C by adding one edge at each step and keeping C connected. Suppose there are k edges in C, and the sequence in which they are added is $e_1, e_2, \cdots, e_k$.

As e_1 is a non-tree edge to T, e_1 and the paths from its two endpoints to the lowest common ancestor (lca) of the two endpoints form a simple cycle. And e_1 is in a biconnected component of G.

Suppose the first l edges in the sequence are in one biconnected component Bc. We now consider adding the $(l+1)^{\text{th}}$ edge. As C is connected, $e_{l+1} = (u, w)$ is adjacent to some edge, say, $e_s = (v, w)$ (where $1 \leq s \leq l$) in the tree we have grown so far at vertex w. By Lemma 1 there are no ancestral relationships between u and w, and v and w in tree T. If there is also no ancestral relationship between u and w as illustrated in part (a) of Fig. 2, then the paths in T from u to $lca(u, v)$ and from v to $lca(u, v)$ plus the edges (u, w) and (v, w) in C form a simple cycle S. As (v, w) is in Bc and (u, w) is in S, and Bc shares with S the edge (v, w), so (u, w) and (v, w) are both in the biconnected component that contains $Bc \cup S$. If there is some ancestral relationship between u and v, then there are two cases: either u is the ancestor of v or v is the ancestor of u. These two cases are illustrated respectively by parts (b) and (c) in Fig. 2. Let's first consider the case that u is the ancestor of v. The paths in T from u to $lca(u, w)$, from w to $lca(u, w)$, and from v to u, and edge (v, w) form a simple cycle S. S shares with Bc edge (v, w), again (u, w) and (v, w) are both in the biconnected component that includes $Bc \cup S$. Similarly we can prove (u, w) and (v, w) are in one biconnected component for the case that v is the ancestor of u. By induction, it follows that all edges of C are in one biconnected component. □

Part (d) of Fig. 2 shows an example that (u, w) and (v, w) are not in one biconnected component if T is not a BFS tree and there are ancestral relationships between u, v, and w.

Theorem 2 *The edges of each connected component of $G - T$ are in one biconnected component.*

Proof: Let C be a connected component of $G - T$. If C is a tree, by Lemma 2, all edges

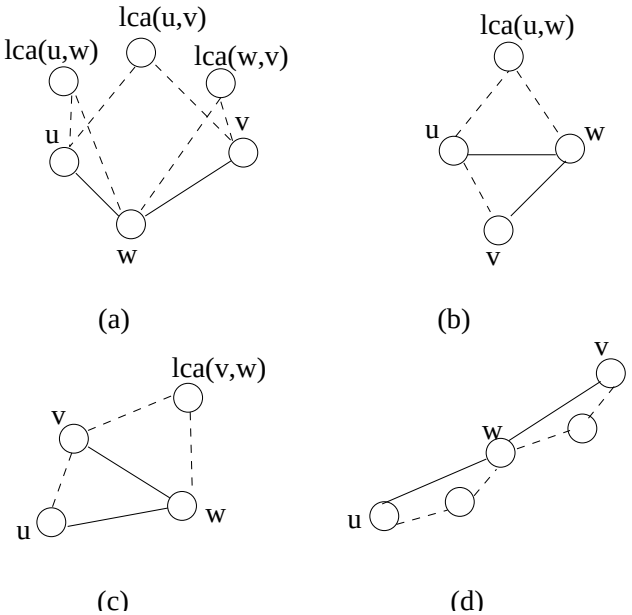

Figure 2: Illustration of the proof of theorem 2. $e_{l+1} = (u, w)$, and $e_s = (v, w)$. The dotted lines are the paths in T while the solid lines are edges in C

of C are in a biconnected component. If C is not a tree, then there exits a spanning tree T_C of C. All edges of T_C are in a biconnected component by Lemma 2. Each nontree edge e (relative to T_C) in C forms a simple cycle with paths in T_C, and the cycle shares the paths with the biconnected component that T_C is in, so e is also in the biconnected component. □

The immediate corollary to Theorem 2 is that we can compute the number of biconnected components in a graph using breadth-first traversal. The first run of BFS computes a rooted spanning tree T. The second run of BFS computes a spanning forest F for $G - T$, and the number of components in F is the number of biconnected components in G.

Next we apply the idea of filtering out *non-essential* edges to the parallel biconnected components problem. First T and F for G are computed. Then biconnected components for $T \cup F$ are computed using a suitable biconnected components algorithm, e.g., the Tarjan-

Vishkin algorithm. We are yet to find the biconnected components to which they belong for all the edges that are filtered out, i.e., edges in $G - (T \cup F)$. According to condition 1 (which holds for arbitrary rooted spanning tree), edge $e = (u, v) \in G - (T \cup F)$ is in the same biconnected component of $(u, p(u))$ if $v < u$. A new algorithm using the filtering technique is shown in Alg. 2.

Input: A connected graph $G = (V, E)$
Output: Biconnected components of G
begin
1. compute a breadth-first search tree T for G;
2. for $G - T$, compute a spanning forest F;
3. invoke TV on $F \cup T$;
4. **for** *each edge* $e = (u, v) \in G - (F \cup T)$ **do**
   label e to be in the biconnected component that contains $(u, p(u))$;

end

Algorithm 2: An improved algorithm for biconnected components.

In Alg. 2, step 1 takes $O(d)$ time with $O(n)$ processors on arbitrary CRCW PRAM where d is the diameter of the graph; step 2 can be done in $O(\log n)$ time with $O(n)$ processors on arbitrary CRCW PRAM [20]; step 3 is the Tarjan-Vishkin algorithm which can be done in $O(\log n)$ time with $O(n)$ processors; finally, step 4 can be implemented in $O(1)$ time with $O(n)$ processors. So Alg. 2 runs in $O(d + \log n)$ time with $O(n)$ processors on CRCW PRAM.

Asymptotically, the new algorithm does not improve the complexity bound of TV. In practice, however, step 2 filters out at least $\max(m - 2(n - 1), 0)$ edges. The denser the graph becomes, the more edges are filtered out. This can greatly speedup the execution of step 3. Recall that TV inspects each nontree edge to compute the *low* and *high* values for the vertices, and builds an auxiliary graph with the number of vertices equal to the number of edges in G. In Section 5 we demonstrate the efficiency of this edge filtering technique.

For very sparse graphs, d can be greater than $O(\log n)$ and becomes the dominating factor in the running time of the algorithm. One pathological case is that G is a chain ($d = O(n)$), and computing the BFS tree takes $O(n)$ time. However, pathological cases are rare. Palmer [15] proved that almost all random graphs have diameter two. And even if $d > \log n$, in many cases, as long as the number of vertices in the BFS frontier is greater than the number of processors employed, the algorithm will perform well on a machine with p processors ($p \ll n$) with expected running time of $O\left(\frac{n+m}{p}\right)$. Finally, if $m \leq 4n$, we can always fall back to *TV-opt*.

5 Performance Results and Analysis

This section summarizes the experimental results of our implementation. We tested our shared-memory implementation on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache. We implement the algorithms using POSIX threads and software-based barriers.

We test our implementation on arbitrary, sparse inputs which are the most challenging instances for previous experimental studies. We create a random graph of n vertices and m edges by randomly adding m unique edges to the vertex set. The sequential implementation implements Tarjan's algorithm.

Fig. 3 shows the performance of *TV-SMP*, *TV-opt* and *TV-filter* on random graphs of $1M$ vertices with various edge densities. For all the instances, *TV-SMP* does not beat the best sequential implementation even at 12 processors. *TV-opt* takes roughly half the execu-

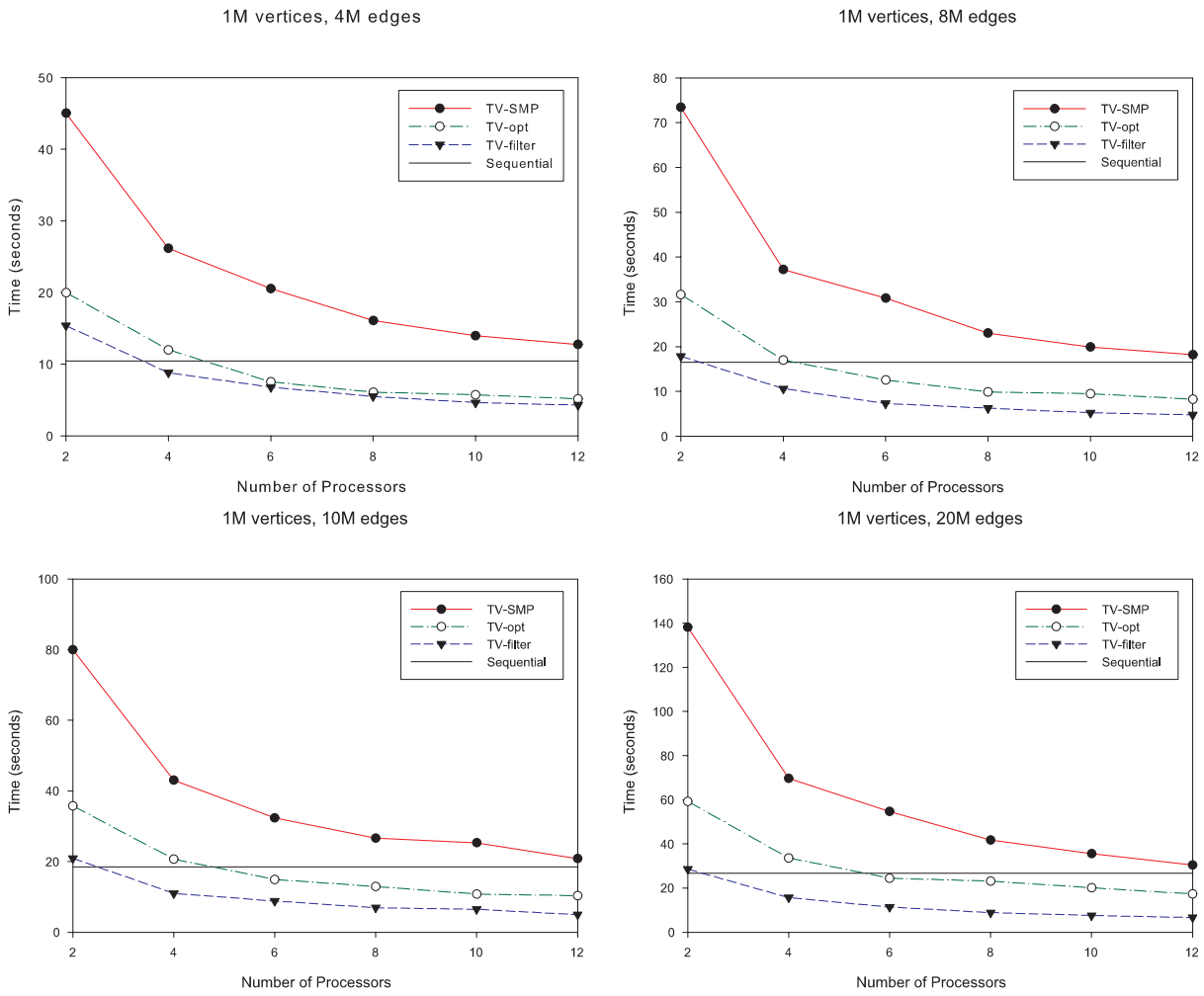

Figure 3: Comparison of the performance of *TV-SMP*, *TV-opt* and *TV-filter* for graphs with $n = 1M$ vertices and various edge densities. "Sequential" is the time taken for the implementation of Tarjan's biconnected components algorithm for the same problem instance.

tion time of *TV-SMP*. As predicted by our analysis earlier, the denser the graph, the better the performance of *TV-filter* compared with *TV-opt*. For the instance with 1M vertices, 20M edges ($m = n \log n$), *TV-filter* is twice as fast as *TV-opt* and achieves speedups up to 4 compared with the best sequential implementation.

Fig. 4 shows the breakdown of execution time for different parts of the algorithm for *TV-SMP*, *TV-opt*, and *TV-filter*. Comparing *TV-SMP* and *TV-opt*, we see that *TV-SMP* takes much more time than *TV-opt* to compute a spanning tree and constructing the Euler-tour.

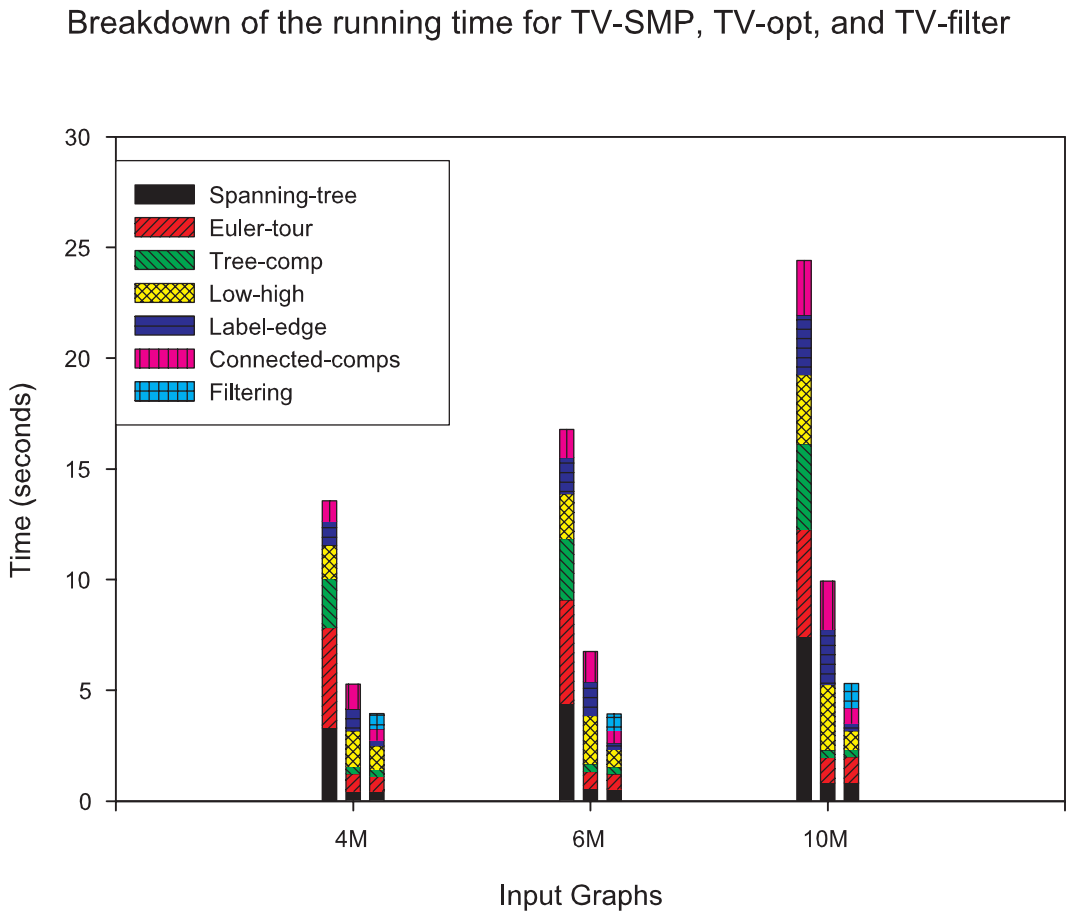

Figure 4: Breakdown of execution time at 12 processors for the *Spanning-tree*, *Euler-tour*, *root*, *Low-high*, *Label-edge*, *Connected-components*, and *Filtering* steps. All graphs have 1M vertices with different number of edges shown on the x-axis. The three columns for each input graph, from left to right, are the execution times for *TV-SMP*, *TV-opt*, and *TV-filter*, respectively.

Also for tree computations, *TV-opt* is much faster than *TV-SMP* because in *TV-opt* prefix sum is used while in *TV-SMP* list ranking is used. For the rest of the computations, *TV-SMP* and *TV-opt* take roughly the same amount of time. Compared with *TV-opt*, *TV-filter* has an extra step, i.e., filtering out *non-essential* edges. The extra cost of filtering out edges is worthwhile if the graph is not extremely sparse. As our analysis predicted in Section 4, we expect reduced execution time for *TV-filter* in computing low-high values, labeling, and computing connected components. Fig. 4 confirms our analysis.

6 Conclusions

We present an experimental study of biconnected components algorithms based on the Tarjan-Vishkin approach on SMPs. Our implementation achieves speedups up to 4 with 12 processors on the Sun E4500. As quite a few fundamental parallel primitives and routines such as prefix sum, list ranking, Euler-tour construction, tree computation, connectivity and spanning tree are employed as building blocks, our study shows optimistic results for parallel algorithms that take drastically different approach than the straightforward sequential approach.

References

[1] B. Awerbuch and Y. Shiloach. New connectivity and MSF algorithms for shuffle-exchange network and PRAM. *IEEE Transactions on Computers*, C-36(10):1258–1263, 1987.

[2] 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.

[3] D. A. Bader and G. Cong. Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[4] D.A. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002. Springer-Verlag.

[5] K.W. Chong and T.W. Lam. Approximating biconnectivity in parallel. *Algorithmica*, 21:395–410, 1998.

[6] G. Cong and D. A. Bader. The Euler tour technique and parallel rooted spanning tree. In *Proc. Int'l Conf. on Parallel Processing (ICPP)*, pages 448–457, Montreal, Canada, August 2004.

[7] D.M. Eckstein. BFS and biconnectivity. Technical Report 79–11, Dept. of Computer Science, Iowa State Univ. of Science and Technology, Ames, Iowa, 1979.

[8] 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.

[9] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

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

[11] T. Hsu and V. Ramachandran. On finding a smallest augmentation to biconnect a graph. *SIAM J. Computing*, 22(5):889–912, 1993.

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

[13] S. Khuller and U. Vishkin. Biconnectivity approximations and graph carvings. *Journal of the ACM*, 41(2):214–235, 1994.

[14] Y. Liang, S.K. Dhall, and S. Lakshmivarahan. Efficient parallel algorithms for finding biconnected components of some intersection graphs. In *Proc. of the 19th Ann. Conf. on Computer Science*, pages 48–52, San Antonio, TX, 1991.

[15] E. Palmer. *Graphical Evolution: An Introduction to the Theory of Random Graphs.* John Wiley & Sons, New York, 1985.

[16] R. Read. Teaching graph theory to a computer. In W. Tutte, editor, *Proc. 3rd Waterloo Conf. on Combinatorics*, pages 161–173, Waterloo, Canada, May 1969.

[17] C. Savage and J. JáJá. Fast, efficient parallel algorithms for some graph problems. *SIAM J. Computing*, 10(4):682–691, 1981.

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

[19] R.E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.

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

[21] Y.H. Tsin and F.Y. Chin. Efficient parallel algorithms for a class of graph theoretic problems. *SIAM J. on Comput.*, 13(3):580–599, 1984.

[22] J. Woo and S. Sahni. Load balancing on a hypercube. In *Proc. 5th Int'l Parallel Processing Symp.*, pages 525–530, Anaheim, CA, April 1991. IEEE Computer Society Press.