Sequential and Vectorized Algorithms for Computing the Connected Components of an Undirected Graph

Robert G. Edwards
Supercomputer Computations Research Institute
Florida State University
Tallahassee, FL 32306
Internet: EDWARDS@VSRGE.SCRI.FSU.EDU

Xiao-Jian Li and Alan D. Sokal
Department of Physics
New York University
4 Washington Place
New York, NY 10003
USA
Internet: LIXJ@ACF4.NYU.EDU, SOKAL@ACF4.NYU.EDU

PACS numbers: ??????

December 18, 1990

Abstract

We study the efficiency of sequential (depth-first-search, breadth-first-search and Fischer-Galler-Hoshen-Kopelman) and vectorizable (Shiloach-Vishkin) algorithms for computing the connected components of an undirected graph. The Shiloach-Vishkin algorithm is implemented as a vectorized program for the Cyber 205 (???). [TIMING COMPARED TO FISCHER-GALLER-HOSHEN-KOPELMAN??] This computation plays a key role in Monte Carlo studies of percolation and in the Swendsen-Wang Monte Carlo algorithm for Potts spin models.

1 Introduction

Computing the connected components of an undirected graph is a classic problem in computer science, with numerous applications [1, 2, 3]. In statistical physics, for example, cluster labelling is the key time-consuming step in Monte Carlo studies of percolation [4, 5] and in algorithms of Swendsen-Wang type for Potts and other spin models [6]. At the end of this Introduction, we explain in more detail the importance of this latter application.

Let $G = (V, E)$ be an undirected graph with n vertices and m edges. Then the well-known "depth-first-search" and "breadth-first-search" algorithms [1, 2, 3] each find the connected components of G in time $O(n + m)$. A very different algorithm was devised by Fischer and Galler [7] (see also [8, 9, 10]) and independently rediscovered (in a slightly more complicated form) by Hoshen and Kopelman [11]. This algorithm has a worst-case running time of order mn [OR IS IT "at least n^2"???], but by a slight modification [8, **?**] the worst-case time can be reduced to $O(n + m \log m)$ or even smaller [though not quite to $O(n + m)$]. [AND IN SOME CASES TO $O(n + m)$, EVEN FOR THE ORIGINAL ALGORITHM?? (see Section 2.4) ?????]. Moreover, for important special cases (e.g. independent percolation on a regular lattice [AND OTHERS???]), it is found empirically [ALSO THEORETICALLY???] to run in time $O(n + m)$. However, all of these algorithms suffer from the defect that they are not vectorizable. This severely limits their application on present-day supercomputers.

Eight years ago, Shiloach and Vishkin [12] proposed an elegant parallel algorithm for computing connected components, which runs in time $O(\log n)$ using $n + 2m$ processors (see [13, **?**] for a review of related work). The important feature of this algorithm is that its parallelism is of the single-instruction-multiple-data (SIMD) type — that is, it is vectorizable! In Section 3 we show explicitly how the Shiloach-Vishkin algorithm can be vectorized for any vector computer with gather/scatter hardware.

In Section 4 we carry out a systematic comparative study of the efficiency of two (???) variants of the Fischer-Galler-Hoshen-Kopelman (FGHK) algorithm versus four (six??) variants of the Shiloach-Vishkin (SV) algorithm, for independent bond percolation on the square lattice at a range of occupation probabilities p. The results of this study are somewhat surprising: although the SV algorithm vectorizes efficiently (a factor-of-?? gain over scalar mode), even the best variant of the SV algorithm performs less than a factor of 2 better than the FGHK algorithm, for most values of p — and in fact it performs slightly worse for p near its critical value 1/2. Thus, the gain due to vectorization only barely compensates the added complexity of the vectorizable algorithm. Of course, this result is both problem-dependent and machine-dependent, so different results could conceivably be obtained on different types of graphs or on machines with different architectures.

Our presentation of the FGHK algorithm is strongly influenced by the beautiful paper of Shiloach and Vishkin [12], which we warmly recommend to the reader. Indeed, one of our goals is to point out the close relation between the FGHK and

Shiloach-Vishkin algorithms — and the important differences.

Before embarking on the details of this problem, it is perhaps worthwhile to explain our motivation for studying it in the first place. (After all, we are physicists, not computer scientists.) The Ising model is probably the single most-studied model in statistical mechanics, of great importance for the theory of critical phenomena and hence indirectly also for quantum field theory. Monte Carlo studies of the Ising model go back to the 1950's [14], and are currently being employed in an attempt to resolve numerically some of the outstanding controversies in the theory of critical phenomena (notably, the hyperscaling conjecture [15, 16]). Since 1980, several (??) tens of person-years of time, and tens [HUNDREDS?] of thousands of dollars, have been expended in the design of special-purpose processors to carry out the standard single-site Metropolis (or heat-bath) algorithm for the Ising model [17, 18, 19, 20, 21]; and a comparable (???) effort has been devoted to vectorizing this same algorithm for general-purpose supercomputers [22, 23, 24, 25, 26, 27]. Very recently, however, Swendsen and Wang [6] have devised an algorithm that is vastly more efficient in the critical region than the Metropolis algorithm, due to its extraordinarily small critical slowing-down. For example, on a 512×512 lattice at the critical point, the autocorrelation time of the Swendsen-Wang algorithm is ≈ 8 [28], while that of the Metropolis algorithm is ≈ 500000 [29]. Thus, even though a sweep of the Swendsen-Wang algorithm takes roughly ??? times the CPU time compared to a sweep of the Metropolis algorithm [refs??? or put also timings in Section 4], the Swendsen-Wang algorithm still emerges as the most efficient known method, by far, for numerical studies of the near-critical ferromagnetic Ising model. The Swendsen-Wang algorithm also applies to the Potts model, and generalizations to nonlinear σ-models have been developed [30, 31]. Since virtually all of the CPU time in the Swendsen-Wang algorithm is spent finding connected clusters [CHECK THIS — CITE EDWARDS-SOKAL — OR PUT IN SECTION 4 TOO??], we think that the desirability of finding improved algorithms for this problem is self-evident.

2 The Algorithms

2.1 Definition of the Problem

Let $G = (V, E)$ be an undirected graph with n vertices, m edges and c connected components. Without loss of generality we can take the vertex set V to be $\{1, 2, \ldots, n\}$.

The graph G may be presented to us in either of two data structures, an *edge list* or an *adjacency list*. An edge list is a sequential listing, in arbitrary order, of the edges of the graph: (i_1, j_1), (i_2, j_2), $\ldots$, (i_m, j_m). The undirected edge $\{i, j\}$ can be listed either as (i, j) or as (j, i). Such a listing obviously takes space $2m$. An adjacency list [**?**] is a collection $\{L_i\}_{1 \le i \le n}$ of linked linear lists, where L_i is a listing, in arbitrary order, of the vertices adjacent to i. Such a list can be stored in a sequentially allocated array, as shown in Figure 1(c): each entry has two fields, a *vertex* field and a *link* field; and we make the convention that linked list L_i starts in location i.[1] [IS THIS HOW MY CCDFS PROGRAM IS WRITTEN?????] This structure takes space $2n + 4m$.

As we shall see, the depth-first-search and breadth-first-search algorithms need their input in the form of an adjacency list, while the Fischer-Galler-Hoshen-Kopelman and Shiloach-Vishkin algorithms need an edge list. It is therefore important to note that either structure can be translated to the other in a time $O(m + n)$. An edge list can be translated to an adjacency list by the following algorithm:

```
for i = 1 to n do
    link(i) ← 0
    tail(i) ← i
enddo
nextnode ← n + 1
for j = 1 to m do
    link(tail(edge1(j))) ← nextnode
    vertex(nextnode) ← edge2(j)
    link(nextnode) ← 0
    tail(edge1(j)) ← nextnode
    nextnode ← nextnode + 1

    [same with edge1 ↔ edge2]

enddo
```

Conversely, an adjacency list can be translated to an edge list by the following algorithm:

```
m ← 0
for i = 1 to n do
    thisnode ← i
    while link(thisnode) ≠ 0
        thisnode ← link(thisnode)
        if vertex(thisnode) > i then
            m ← m + 1
            edge1(m) ← i
            edge2(m) ← vertex(thisnode)
        endif
    endwhile
enddo
```

[1]It is not crucial that L_i start in this particular location, but it *is* crucial that we *know* where L_i starts. If we didn't make this convention, we would need an extra array *head*(i) indicating where L_i starts.

WHAT ABOUT SETTING UP THESE STRUCTURES FOR PERCOLATION? WHICH IS MORE NATURAL/QUICKER??

Finally, it is necessary to specify the data structure in which the algorithm's output is to be presented. For our purposes (e.g. statistical-mechanics applications), the most convenient form is a *cluster labelling*, i.e. a map L: $\{1, \ldots, n\} \to \{1, \ldots, c\}$ such that $L(i) = L(j)$ if and only if i and j are in the same connected component of G. Note that any two cluster labellings differ by a permutation of the labels $\{1, \ldots, c\}$.

2.2 Depth-First Search and Breadth-First Search

The depth-first-search and breadth-first-search algorithms are explained in nearly all textbooks on computer algorithms (e.g. [1, 2, 3, 9, 10]); we therefore given only a very brief summary.

2.3 Preliminaries for the Fischer-Galler-Hoshen Kopelman and Shiloach-Vishkin Algorithms

We begin with a few graph-theoretic definitions: A *rooted tree* is a directed graph in which there exists a (necessarily unique) vertex r, called the *root*, such that:

(a) The underlying undirected graph is a tree (i.e. it is connected and has no cycles).

(b) For each vertex i, there exists a (necessarily unique) directed path from i to r.

The *depth* of a vertex is its distance from the root. The *height* of the tree is the maximum depth of its vertices. A *rooted star* is a rooted tree of height 0 or 1 (i.e. one in which every vertex is adjacent to the root). A *forest of rooted trees (or stars)* is a directed graph, not necessarily connected, in which each connected component is a rooted tree (or star). If i is a vertex in such a forest, we let $T(i)$ be the tree containing i, and $root(i)$ its root.

In any forest of rooted trees, each vertex has out-degree either 0 or 1, depending on whether it is or is not a root. If i is not a root, we let $P(i)$ be the unique vertex adjacent from i; it is called the *parent* of i. If i is a root, then $P(i)$ is undefined. A forest of rooted trees can be conveniently stored as an array of pointers from each vertex to its parent, i.e. an array $P = \{P(i)\}_{1 \le i \le n}$. The roots must, of course, be indicated by some special convention, for which there are various possibilities:

(a) $P(i) = i$. In this case one can alternatively imagine the graph to be a forest of rooted trees *with self-loop at the root*.

(b) $P(i) = 0$.

(c) $P(i) < 0$. In this case one could use the value $-P(i)$ to hold additional information about the tree, e.g. its height or the number of vertices it contains.

The root of the tree containing i can be found by a straightforward search using the pointer array. For example, if roots are represented by $P(i) = i$, the function would be:

> **function** $root(i)$
>
> $itemp \leftarrow i$
> **while** $P(itemp) \neq itemp$
> $itemp \leftarrow P(itemp)$
> **endwhile**
> $root(i) \leftarrow itemp$
> **return**

This computation takes a time proportional to $depth(i) + 1$.

A central role in both the Fischer-Galler-Hoshen-Kopelman and Shiloach-Vishkin algorithms will be played by an auxiliary directed graph $R = (V, F)$ that will evolve as the algorithm proceeds; we call this graph the *pointer graph*. At all times the pointer graph will be a forest of rooted trees. Initially it is a forest of singletons, i.e. each vertex is a root. As the algorithm proceeds, the trees in this forest coalesce by a sequence of operations of "hooking" and "shortcutting" (to be described below). When the algorithm terminates, the pointer graph is a forest of rooted trees (rooted stars in the SV case), each one of which corresponds precisely to a connected component of the original graph G.

For future reference, we let $\widetilde{E}$ denote the set of $2m$ directed edges corresponding to the edges E in both orientations.

2.4 The Fischer-Galler-Hoshen-Kopelman Algorithm

In this section we present the Fischer-Galler [7, 8] [9, pp. 129–145, 165, 169???] [10, pp. 70–79, 94???] version of the FGHK algorithm, together with some of the improvements that have been devised by computer scientists over the past 15 years [?]. At the end of this section, we shall make some brief remarks about Hoshen-Kopelman [11] version, which is perhaps better known to physicists.[2]

The FGHK algorithm begins as follows:

> *Step 0.* Initialize to a forest of singletons.
>   **for each** $i \in V$, **do** $P(i) \leftarrow i$;

[2] We cannot resist making here a personal remark. In the fall of 1987, inspired by Wang's implementation of the Hoshen-Kopelman algorithm [32] and by the Shiloach-Vishkin paper [12], one of us (A.D.S.) invented a generalized (to arbitrary undirected graphs) and streamlined version of the Hoshen-Kopelman algorithm — and was rather proud of himself for doing so. But in May 1988, we learned from Tony Kennedy that this algorithm had in fact been invented nearly a quarter of a century earlier! We think that this little story underlines the need for better communication between physicists and computer scientists.

Step 1. Main loop — hook trees together.

for each $\{i, j\} \in E$, **do sequentially** (in any order)
  if $root(i) < root(j)$ **then**
    $P(root(j)) \leftarrow root(i)$
    $P(i) \leftarrow root(i)$ (*)
    $P(j) \leftarrow root(i)$ (*)
  else
    $P(root(i)) \leftarrow root(j)$
    $P(i) \leftarrow root(j)$ (*)
    $P(j) \leftarrow root(j)$ (*)
  endif
enddo

Thus, in Step 1 we consider in turn each edge $\{i, j\}$ of the graph G, and amalgamate the trees $T(i)$ and $T(j)$ in the pointer graph (if they are not already the same tree) by hooking the higher-numbered of the two roots onto the lower-numbered. The operations marked (*) are optional: their aim is to save time when examining subsequent edges, by shortening the paths from i and j to their root. (In Section 4 we show that the FGHK algorithm runs about 5–10% faster if these optional operations are omitted, at least for the class of graphs we consider.) We use the words "**do sequentially**" to emphasize that the edges must be considered *one after another* (though in arbitrary order) — this is in contrast with the Shiloach-Vishkin algorithm, where the analogous operations may be carried out in parallel.

It is easy to prove, by induction, that at the end of each pass through the **do** loop in Step 1, the pointer graph has the following properties:

(a) It is a forest of rooted trees.

(b) If the edge $\{i, j\}$ has already been examined, then i and j belong to the same tree.

(c) $P(i) \leq i$ for all i, with equality if and only if i is a root.

In particular, at the conclusion of Step 1, the pointer graph is a forest of rooted trees, each tree of which corresponds precisely to a connected component of the graph G; and moreover, $P(i) \leq i$ for all i, so that the root of each tree is its lowest-numbered vertex. Because of this latter property, we can renumber the connected components in a single pass through the vertices:

Step 2. Clean-up — give each vertex its label.

$index \leftarrow 0$
for $i = 1$ **to** n, **do sequentially** (*in this order*)
  if $P(i) = i$ **then**
    $index \leftarrow index + 1$
    $P(i) \leftarrow index$

else
    $P(i) \leftarrow P(P(i))$
endif
enddo

By visiting the vertices in ascending order, the root of each tree will be found first, and can be given its *index*. Moreover, by the time each subsequent vertex in this tree is visited, its parent will already have been visited [since $P(i) \leq i$] and hence also received its *index*. The beauty of this procedure is that it requires no additional storage: the single array P serves as the array of labels for those vertices which have already been visited in the clean-up pass, and as the array of pointers for those vertices which have not yet been visited in the clean-up pass.

It has been found empirically in studies of independent percolation on regular lattices [**?**] (see also Section 4) that the running time of the FGHK algorithm is very closely proportional to the volume of the system, *i.e.* it is of order $n + m$. [DOES THIS DEPEND SENSITIVELY ON THE ORDERING OF THE EDGES?? WHAT IF WE PERMUTED THE EDGES RANDOMLY, RIGHT BEFORE CALLING CCFGHK?? TRY IT!!] However, it is worth noting that for general graphs, the worst-case behavior can be very bad:

Example 1. For any n, there exists a connected graph with n vertices and $n - 1$ edges, and an ordering of the edges, such that the FGHK algorithm (either with or without the optional operations in Step 1) has a running time of order n^2.
Proof. Set $k = lcn/2rc$, and consider the tree graph shown in Figure 1(a), with the vertices and edges numbered as indicated. After the first $k - 1$ edges have been examined, the pointer graph will be as shown in Figure 1(b). The next edge (*i.e.* $\{k,\, k+1\}$) requires a search of $k-1$ steps to find $root(k) = 1$. The edge after that (*i.e.* $\{k-1,\, k+2\}$) requires a search of $k - 2$ steps to find $root(k-1) = 1$, and so on. The total work is thus of order n^2.

Example (Conjecture!!!) 2. For any n and any m satisfying ?????????, there exists a connected graph with n vertices and m edges, and an ordering of the edges, such that the FGHK algorithm (either with or without the optional operations in Step 1???) has a running time of order mn. [IS THIS TRUE???]

The foregoing examples use a rather unnatural ordering of the edges — one virtually opposite to the ordering of the vertices. One might conjecture, therefore, that these pathologies could be avoided by ordering the edges *lexicographically*, that is, by processing first the edges $\{1,\, j\}$ in ascending order of j, then the edges $\{2,\, j\}$ with $j > 2$ in ascending order of j, and so on. This conjecture is, however, false:
Example 3. For any n, there exists a connected graph with n vertices and $n - 1$ edges, such that the FGHK algorithm (either with or without the optional operations in Step 1) with lexicographic ordering of the edges has a running time of order n^2.
Proof. Assume for simplicity that $n = 5k$ with integer k, and consider the graph shown in Figure 2(a), with the vertices numbered as indicated. After all edges $\{i, j\}$

with $i \leq k$ have been processed, the pointer graph is as shown in Figure 2(b). Next process the edge $\{k+2,\, k+3\}$; the resulting pointer graph is shown in Figure 2(c). Next process the edges $\{k+4,\, k+5\}$, $\{k+6,\, k+7\}$ and so on. After all edges $\{i, j\}$ with $i \leq 3k$ have been processed, the pointer graph is as shown in Figure 2(d): the vertex 1 is a root, and the vertices 2, ..., k are at heights [DEPTHS??] 1, ..., $k-1$ in that same tree. Finally, process the k remaining edges (in any order); the searches for their roots will require 1, 2, ..., k steps. So the total run-time is of order n^2.

If n is not a multiple of 5, set $k =???n/5???$ and delete from Figure 2(a) those vertices numbered n. The run-time is still of order n^2.

[NOW COMPARISON TO HK VERSION OF FGHK ALGORITHM AND DISCUSSION] Cite various published implementations of HK algorithm: Binder, Stauffer and their friends; D.C. Rapaport, Comput. Phys. Reports **5**, 265 (1987), section 7.

2.5 The Shiloach-Vishkin Algorithm

An extraordinarily clear exposition of the Shiloach-Vishkin algorithm is given in the original paper [12], which we recommend to the reader. In order to make the present paper self-contained, we give here a brief description of the algorithm. [OUR VERSION IS SLIGHTLY DIFFERENT FROM THE ORIGINAL IN THAT IT REQUIRES LESS SYNCHRONIZATION — MAKE SURE CORRECTNESS PROOF STILL HOLDS!!!]

The Shiloach-Vishkin algorithm uses a model of synchronized parallel computation with shared memory, in which simultaneous writing into the same memory location is permitted. Each block of code marked "**do in parallel**" is begun and ended by all processors simultaneously, *i.e.* no processor goes on to a subsequent section of code until all processors have finished the current block. However, the relative order, between different processors, of instruction executions within a given "**do in parallel**" block is uncontrollable. This has the following consequences:

(a) If a processor tries to read a memory location which is being written on by another processor during the same block, it may obtain either the "old" or the "new" value.

(b) If two or more processors try to write on the same memory location during the same block, one of them will succeed, but we do not know which one.

Algorithms for such a machine must have the property that they give the correct output (and in the claimed maximum time, if any) for *all* admissible orderings of the instruction executions.

The basic idea of the Shiloach-Vishkin algorithm is very similar to that of the FGHK algorithm: the trees of the pointer graph coalesce by repeated "hooking" operations until each one corresponds precisely to a connected component of the original graph G. However, because the "hooking" operations for the different edges

$\{i, j\}$ are carried out in parallel rather than sequentially, the algorithm may not reach its final state in a single iteration. (The trouble is that several different edges $\{i, j\}$ may try simultaneously to hook the same root $r = root(i)$ onto different vertices, and only one of them will succeed.) Thus, the basic "hooking" step may have to be repeated a number of times in order for the final state to be reached. In addition, a "shortcutting" operation is employed to cut down the height of the trees in the pointer graph.

For future reference, we note that all steps in the Shiloach-Vishkin algorithm have the property that roots may be destroyed but never created; that is, once a vertex loses its property of being a root, it will never again become a root.

The SV algorithm begins as follows:

Step 0. Initialize to a forest of pure self-loops. Initialize stagnancy flags to zero.

$s \leftarrow 1$
for each $i \in V$, **do in parallel**
    $P_{prev}(i) \leftarrow i$
    $Q(i) \leftarrow 0$
enddo

Comment. s is the iteration counter; P is the pointer array for the current iteration; P_{prev} is the pointer array as of the end of the previous iteration. The role of the array Q will be explained later.

Step 1. First shortcutting.

iflag $\leftarrow 0$
for each $i \in V$, **do in parallel**
    $P(i) \leftarrow P_{prev}(P_{prev}(i))$
enddo

Comment. *iflag* is set to zero at the beginning of each iteration, and becomes nonzero if the pointer graph is changed (see Step 3). The "shortcutting" step transforms each tree of the pointer graph into a new tree with the same root and smaller height: to be precise, tree of height h is transformed into one of height $\lceil h/2 \rceil$.

Step 2. Hook trees together.

for each $(i, j) \in \widetilde{E}$, **do in parallel**
    if $P(i) = P_{prev}(i)$ and $P(j) < P(i)$ **then**
        $P(P(i)) \leftarrow P(j)$
    endif
enddo

Comment. The "hooking" step is as follows: each vertex i which pointed to a root at the end of the previous iteration $[P(i) = P_{prev}(i)]$ checks whether any of its neighbors j are pointing to lower-numbered vertices $[P(j) < P(i)]$. If so, an attempt is made to hook the tree $T(i)$ [whose root is $P(i)$] onto $P(j)$. Of course, several different "children" i of the same root $P(i)$, and several different neighbors j of i, may try simultaneously to hook $T(i)$ to different places. One will succeed; we do not know (and do not care) which one. We note also that because the edges (i, j) are being processed *in parallel*, the values for $P(i)$ and $P(j)$ used in the **if** and the **then** steps may be either the "old" or the "new" values. [IS THIS WHAT WE WANT, AT LEAST AS REGARDS THE "IF" TEST???]

At this point we need a definition: a tree is said to be *stagnant in the s^{th} iteration* if it has not been changed in the first two Steps of this iteration, i.e. it has not been shortcut, no tree has been hooked onto it, and it has not been hooked onto any other tree. A root of a stagnant tree is a *stagnant root*. Equivalently, a stagnant root is one which is still a root after Step 2, and which has no new vertices pointing to it that did not already point to it at the end of the previous iteration. (Note that if in Step 2 something got hooked onto a vertex of the tree other than the root, then the tree must have height ≥ 2, in which case the root became non-stagnant also in Step 1.)

The role of the array Q is to mark stagnant roots: in Step 3 of the s^{th} iteration, Q will be set so that $Q(i) = s$ if after Step 2 of the s^{th} iteration there exists at least one vertex j pointing to i that did not point to i after the $(s-1)^{st}$ iteration, and $Q(i) < s$ otherwise. Then, in Step 4, each vertex i that points to a stagnant root $[P(i) = P(P(i))$ and $Q(P(i)) < s]$ checks whether any of its neighbors j is pointing to a vertex of another tree $[P(j) \neq P(i)]$. If so, an attempt is made to hook the tree $T(i)$ [whose root is $P(i)$] onto $P(j)$. Of course, several different "children" i of the same stagnant root $P(i)$, and several different neighbors j of i, may try simultaneously to hook $T(i)$ to different places. One will succeed; we do not know (and do not care) which one. The purpose of Steps 3 and 4 is explained below.

Step 3. Mark non-stagnant vertices.

```
for each i ∈ V, do in parallel
    if P(i) ≠ P_prev(i) then
        Q(P(i)) ← s
        iflag ← 1
    endif
enddo
```

Step 4. Hook stagnant trees.

```
for each (i, j) ∈ Ẽ, do in parallel
    if P(i) = P(P(i)) and Q(P(i)) < s and P(j) ≠ P(i) then
        P(P(i)) ← P(j)
    endif
enddo
```

Comment. Note that because the edges (i, j) are being processed *in parallel*, the values for $P(i)$, $P(P(i))$ and $P(j)$ used in the **if** and the **then** steps may be either the "old" or the "new" values. [IS THIS WHAT WE WANT, AT LEAST AS REGARDS THE "IF" TEST???]

Step 5. Second shortcutting.
for each $i \in V$, **do in parallel**
$P_{prev}(i) \leftarrow P(P(i))$
enddo

Comment. The result of this step goes in P_{prev}, since that is where it will be needed in Step 1 of the next iteration.

Step 6. Test for completion of main iteration.
if *iflag* $\neq 0$ **then** go to Step 1

Comment. *iflag* $= 0$ means that the pointer graph was not changed during the first two steps of the s^{th} iteration. In this case it can be proven [12] that nothing was changed during Step 4 or 5 either (hence $P = P_{prev}$), and that the pointer graph is a forest of rooted stars with self-loop, each one corresponding precisely to a connected component of the original graph G. This means that we are ready to begin the clean-up phase.

Step 7. Clean-up: find the roots and number them sequentially.
$index \leftarrow 0$
for each $i \in V$, **do sequentially** (in any order)
if $P(i) = i$ **then**
$index \leftarrow index + 1$
$L(i) \leftarrow index$
endif
enddo

Comment. This step will not parallelize, but it does vectorize on any machine with a COMPRESS or EXPAND instruction. We remark that the array L can be stored in the same location as Q, since the latter is no longer needed. [WAIT A MINUTE: IS IT TRUE THAT $P(i) \leq i$ FOR ALL i??? IF IT IS, THEN WE CAN DO THE CLEAN UP IN A SINGLE PASS, AS IN THE FGHK ALGORITHM.]

Step 8. Clean-up: give each vertex its label.
for each $i \in V$, **do in parallel**
$L(i) \leftarrow L(P(i))$
enddo

Comment. $L(i)$ is now the label of vertex i.

Shiloach and Vishkin [12] have proven that for any admissible ordering of the instruction executions, the algorithm terminates after at most ???$\log_{3/2} n$??? + 2 iterations, with $L(i) = L(j)$ if and only if i and j are in the same connected component of G. In other words:

(a) The output of the algorithm (array L) does not depend on the order of instruction executions within each Step, except for a possible permutation of the cluster labels.

(b) The running time of the algorithm may depend on the order of instruction executions, but it is never worse than ???$\log_{3/2} n$??? + 2 iterations.

In particular, the algorithm works correctly on a scalar (sequential) machine.

We now make some remarks concerning variants of the SV algorithm:

1) The purpose of singling out stagnant roots for special treatment (Steps 3 and 4) is to avoid certain situations in which the number of iterations can be of order n rather than $\log n$. One example, taken from [12], is shown in Figure 3 (#??); this example shows that if Steps 3 and 4 are omitted, the *worst-case* running time (for certain orderings of the instruction executions) can be of order n. It might nevertheless be thought that such situations are "rare", and that the *average* running time could be reduced (perhaps by a factor of 2) by omitting Steps 3 and 4 (except for the setting of *iflag* in Step 3). We call this *modification A* of the SV algorithm. In Section 4 we test its performance.

2) Step 5 (the second shortcutting) is optional, and could be omitted. On the other hand, one might consider *strengthening* Step 5 so as to cut down more rapidly the height of the trees in the pointer graph. (Since the goal of the algorithm is to obtain a forest of rooted stars, one would expect that any action to cut down the height of trees would reduce the number of iterations required. If this is the case, then the question is whether this reduction justifies the extra cost of a strengthened shortcutting operation.) There is a hierarchy of strengthened shortcutting operations, the strongest of which is the "complete shortcutting" $P(i),\ P_{prev}(i) \leftarrow root(i)$:

Step 5^∞. "Complete shortcutting": make the pointer graph into a forest of rooted stars.
    while there exists $i \in V$ such that $P(P(i)) \neq P(i)$
        for each $i \in V$, **do in parallel** $P_{prev}(i) \leftarrow P(P(i))$;
        for each $i \in V$, **do in parallel** $P(i) \leftarrow P_{prev}(i)$;
    endwhile

Note also that if Step 5^∞ is used in place of Step 5, then Step 1 can be omitted, as can the test "**if** $P(i) = P_{prev}(i)$" in Step 2. (Step 0 should then initialize both P and P_{prev}.) We call this *modification B* of the SV algorithm, and in Section 4 we test its performance.

3) In Steps 2 and 4 of the SV algorithm, we always hook the vertex $P(i)$ onto the parent of i's neighbor, $P(j)$. It might, however, be advantageous to hook it higher up on the tree $T(j)$ [*e.g.* to the "grandparent" $P(P(j))$], so as to reduce the height of the amalgamated tree. [Note that in the FGHK algorithm it is always hooked onto $root(j)$.] In general this would involve a tradeoff between a possible reduction in the number of iterations versus an increase in the computation required per iteration. However, in one case we can make a modification of this type "for free": in Step 3, we have already computed the array $P(P(\cdot))$ in order to perform the **if** test, so we can replace the instruction $P(P(i)) \leftarrow P(j)$ in Step 4 (???) by $P(P(i)) \leftarrow P(P(j))$ at no cost whatsoever. We call this *modification C* of the SV algorithm, and in Section 4 we test whether it provides any significant benefit.

The above three modifications of the SV algorithm can be employed in combination, except that modifications A and C are incompatible. We thus have six versions of the SV algorithm: 0 (original), A, B, C, AB, BC.

3 The Programs

3.1 (Scalar) Implementation of the Fischer-Galler-Hoshen-Kopelman Algorithm

In Table 1 we give a standard FORTRAN 77 implementation of the FGHK algorithm. This code is a straightforward transcription of the algorithm as explained in Section [**?**]. The only points to notice are ...????

3.2 Scalar Implementation of the Shiloach-Vishkin Algorithm

In Table 2 we give a standard FORTRAN 77 implementation of the Shiloach-Vishkin algorithm. This code is a straightforward transcription of the algorithm as explained in Section 2.5. The only points to notice are ... ???

3.3 Vectorized Implementation of the Shiloach-Vishkin Algorithm

In Table 3 we give a vectorized implementation of the Shiloach-Vishkin algorithm, in standard FORTRAN 77, that should run on any vector computer with gather/scatter hardware and a sufficiently smart compiler. We have run on the ETA-10 using the VAST-2 version ??? pre-processor, and on the Convex C-210 using the ???? compiler. .... ???

4 Tests and Timings

[EXPLAIN TESTS MADE TO VERIFY CORRECTNESS OF ALGORITHM IN BOTH SCALAR AND VECTOR FORMS]

[GIVE TIMINGS OF SV vs. HOSHEN-KOPELMAN (vs. depth-first-search?) ON VAX 8650 AS A FUNCTION OF LATTICE SIZE; GIVE TIMINGS OF SCALAR

SV vs. VECTORIZED SV vs. HOSHEN-KOPELMAN (vs. depth-first-search?) ON CYBER 205 AS A FUNCTION OF LATTICE SIZE]

5 Discussion

Discuss generalizations: biconnected components, strongly connected components, other graph algorithms ...

Discuss parallel algorithms for specific architectures (eg. hypercube): recently published paper; explain Brower-Tamayo method

Discuss importance of gather/scatter hardware: cite J.G. Lewis and H.D. Simon, SIAM J. Sci. Stat. Comput. **9**, 304 (1988); A. Berretti and A.D. Sokal, Comput. Phys. Commun. **??**, ??? (1990??); and ask vectorization experts at Courant for other references.

Acknowledgments

We would like to thank Ulli Wolff for emphasizing to us the importance of this problem, Mal Kalos for pointing out the work of Shiloach and Vishkin, Tony Kennedy for pointing out the work of Fischer and Galler, and Larry Rudsinksi for help in vectorizing the SV algorithm for the Cyber 205. These computations were carried out on the VAX 8650 and Convex C-210 at the New York University Academic Computing Facility, and on the Cyber 205 and ETA-10 at the John Von Neumann Supercomputer Center. This work was supported in part by NSF grants PHY-8413569 and DMS-8705599 and JVNC grant NAC-705.

References

[1] E.M. Reingold, J. Nievergelt and N. Deo, *Combinatorial Algorithms: Theory and Practice* (Prentice-Hall, Englewood Cliffs NJ, 1977), Chapter 8.

[2] A. Gibbons, *Algorithmic Graph Theory* (Cambridge University Press, 1985), Chapter 1.

[3] S. Even, *Graph Algorithms* (Computer Science Press, Potomac MD, 1979), Chapter 3.

[4] K. Binder and D. Stauffer, in *Applications of the Monte Carlo Method in Statistical Physics*, ed. K. Binder (Springer-Verlag, Berlin, 1984), section 8.4.

[5] D. Stauffer, *Introduction to Percolation Theory* (Taylor & Francis, London, 1985).

[6] R.H. Swendsen and J.-S. Wang, Phys. Rev. Lett. **58**, 86 (1987).

[7] B.A. Galler and M.J. Fischer, Commun. ACM **7**, 301, 506 (1964).

[8] D.E. Knuth, *The Art of Computer Programming*, vol. 1, 2^{nd} ed. (Addison-Wesley, Reading, Massachusetts, 1973), pp. 353-355, 360, 572.

[9] A.V. Aho, J.E. Hopcroft and J.D. Ullman, *The Design and Analysis of Computer Algorithms* (Addison-Wesley, Reading, Massachusetts, 1974).

[10] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms* (Computer Science Press, Rockville, Maryland, 1978).

[11] J. Hoshen and R. Kopelman, Phys. Rev. **B14**, 3438 (1976).

[12] Y. Shiloach and U. Vishkin, J. Algorithms **3**, 57 (1982).

[13] M.J. Quinn and N. Deo, Computing Surveys **16**, 319 (1984).

[14] L.D. Fosdick, in *Methods of Computational Physics*, vol. 1, ed. B. Alder, S. Fernbach and M. Rotenberg (Academic Press, New York, 1963), pp. 245-280.

[15] M.E. Fisher and J.-H. Chen, J. Physique **46**, 1645 (1985).

[16] A.J. Guttmann, Phys. Rev. **B33**, 5089 (1986).

[17] A. Hoogland, J. Spaa, B. Selman and A. Compagner, J. Comput. Phys. **51**, 250 (1983)

[18] R.B. Pearson, J.L. Richardson and D. Toussaint, J. Comput. Phys. **51**, 241 (1983).

[19] J.H. Condon and A.T. Ogielski, Rev. Sci. Instr. **56**, 1691 (1985).

[20] H.J. Herrmann, Physica **140A**, 421 (1986).

[21] M. Taiji, N. Ito and M. Suzuki, Rev. Sci. Instr. **???**, ??? (1988?? 89??).

[22] S. Wansleben, J.G. Zabolitzky and C. Kalle, J. Stat. Phys. **37**, 271 (1984).

[23] C. Kalle, J. Phys. **A17**, L801 (1984).

[24] S.F. Reddaway, D.M. Scott and K.A. Smith, Comput. Phys. Commun. **37**, 351 (1985).

[25] M. Kikuchi and Y. Okabe, Progr. Theor. Phys. **74**, 458 (1985).

[26] G. Bhanot, D. Duke and R. Salvador, J. Stat. Phys. **44**, 985 (1986).

[27] N. Ito and Y. Kanada, Supercomputer **5**, 31 (1988).

[28] R.G. Edwards, X.-J. Li and A.D. Sokal, in preparation. Some preliminary data are found in A.D. Sokal, in *Computer Simulation Studies in Condensed Matter Physics: Recent Developments* (Athens, Georgia, 15–26 February 1988), ed. D.P. Landau, K.K. Mon and H.-B. Schüttler (Springer-Verlag, Berlin–Heidelberg, 1988).

[29] N. Ito, M. Taiji and M. Suzuki, J. Physique (Suppl.) **49**, C8–1397 (1988).

[30] U. Wolff, Phys. Rev. Lett. **62**, 361 (1989); Nucl. Phys. **B322**, 759 (1989).

[31] R.G. Edwards and A.D. Sokal, Phys. Rev. **D40**, 1374 (1989).

[32] J.-S. Wang, private communication.

PUT GRAPH HERE!!!

	edge1	*edge2*
1	1	3
2	1	2
3	2	3
4	3	4

	vertex	*link*
1		6
2		9
3		7
4		13
5		0
6	3	8
7	1	11
8	2	0
9	1	10
10	3	0
11	2	12
12	4	0
13	3	0

Figure 1: (a) An undirected graph. (b) The edge list. (c) The adjacency list.