Chapter Seven

Connected Component Labelling in Image Processing with MIMD Architectures

R. Hummel

1 INTRODUCTION

Many low-level vision tasks are ideally suited for parallel computers configured as a large grid of locally connected simple processors executing a single instruction stream. Such SIMD (Single Instruction, Multiple Data) image-processing computers now exist [1–3]. Common tasks such as point operations, thresholding, convolution, texture analysis and median filtering are easily transferred to such an architecture. A range of other tasks, such as morphological operations (shrinking, expanding, smoothing), region growing, and relaxation operations for segmentation are also frequently well suited to mesh arrays. However, many of these operations are really ways of identifying structure within the image, and to make measurements in particular locations within the image, by treating each pixel in a uniform way. That is, on a mesh-connected architecture, these algorithms in fact perform useful work on only a fraction of the image pixels. Finally, for many higher-level vision tasks, such as boundary parameterization, feature extraction, object matching and symbolic constraint propagation, mesh-connected architectures can at best clumsily provide the necessary data-communication paths.

Some of the mesh-connected machines are enhanced with global operations, such as a "sum-OR" or even a global sum of bits, in order to give the machine the ability to perform fast image maxima or histograms. This increases the ability to extract Hough transforms and other image-wide features. The difficulty here is that region-based features and local Hough

transforms can only be accomplished by masking out the relevant regions, and operating serially on the individual regions.

Certain algorithms are necessarily nonlocal. For example, connected-component extraction and convex-hull operations on binary images require long-range communication, as pointed out in [4]. Although these can be implemented on mesh-connected machines, typically a large number of processing steps are required. For example, for connected-component labelling, the standard algorithm is equivalent to a breadth-first-search, and works by having pixels within each region iteratively replace a current identifier with the maximum of identifiers within its local region. If each pixel begins with a unique identifier, then eventually each component will be labelled with the number representing the maximum identifier in that component at the start of the process. An asymptotically faster approach is available, which works by a divide-and-conquer approach [5]. Connected-component analysis is done within each of four quadrants of the image, by a recursive procedure, and then the components are combined by an equivalence table analysis, done within the mesh, along the interface borders of the quadrants. This latter algorithm can work in time $O(n)$ on an n by n grid. On a pyramid machine, the worst-case complexity can be further reduced to $O(n^{\frac{1}{2}})$ [6].

Once components are extracted, it is usual in image processing to analyse those components, and form complex representations, in order to perform recognition and scene analysis. Features like moments, Fourier descriptors of the boundary, other curvature measurements, three-dimensional shape analysis, and graph structure of regions generally argue for a standard, powerful processor capable of accessing all the image data. Parallel architecture design would then suggest that if one processor with global access is useful for higher-level vision analysis, then two processors will be more advantageous. In fact, if the standard image has a few dozen or a few hundred regions whose shapes, descriptions and local context require analysis in relative isolation from the remaining portions of the image, we would like to have available several dozen or hundreds of processors, each capable of running an independent instruction stream, and each reconfigurable in the sense that it can be assigned to a variable portion of the image. We are thus led to consider MIMD (Multiple Instruction, Multiple Data) shared-memory architectures for image processing.

Parallelism of the order of a few hundred or thousand processors in MIMD shared memory mode offers a number of attractive features for image processing. Since the architecture can be considered "medium-grained" (granularity is relative; here we are comparing with the number of pixels in a typical image), each processor can be a quite capable multi-MIPS (Million Instructions per Second) microprocessor, with a rather complete

instruction set. In particular, we see each processor as having registers, a local cache memory, as well as complete access to global memory. This compares with fine-grain, typical SIMD architectures, where the processors have access exclusively to a few hundred bytes of local memory, and are typically bit-serial or 8-bit processors with an extremely limited instruction set. The use of powerful processors eases algorithm development and enhances flexibility in design and coding. We should expect that nearly all standard image processing algorithms can be handled with relatively good efficiency and programming ease on a multiprocessor shared-memory design.

Further, reconfigurability of the assignment of processors to image regions coincides with notions of "focus of attention" and foveal resolution versus periphery processing. Each processor can work on a specific component, communicating with neighbouring components through the relevant processors by means of an "edge adjacency list" data structure for the dynamic graph of image regions.

Finally, since processor power can be targetted to relevant portions of the image, we expect to increase the degree of parallelization. In grid arrays, with fixed assignment of processors to pixels, many of the processors perform no useful work during execution of a cycle, simply because they are statically assigned to locations where work is not needed. Thus many algorithms yield degrees of parallelization in the range of a fraction of a percent. With the reconfigurability of a medium-grain architecture, we expect to increase the usage figure, at least for higher-level image processing tasks, to the tens of percent.

There are now a number of proposed and prototype MIMD shared-memory computers [7,8]. The use of these machines for image processing has received some attention [9,10]. However, the conversion of standard algorithms to MIMD parallel code offers some subtleties. Very high-level vision tasks presumably offer few obstacles, because little coordination between processors is necessary. Sharing of data is usually minimal. However, for lower-level vision tasks, there are difficulties related to synchronization and assignment of processors. In this paper, we study as an example case the graph-theory algorithm of Shiloach and Vishkin for connected component labelling [11]. This algorithm, like most intermediate-level vision tasks, is presented as an SIMD parallel algorithm. We study how such an algorithm can be mapped to a realistic MIMD machine. Our model for the MIMD machine is the NYU Ultracomputer [8]. Our choice of the connected component algorithm is based on the observations that component labelling is necessarily a non-local operation and that the $O(\log N)$ algorithm is non-obvious but easily described and coded.

There are two main concerns that arise in converting SIMD parallel

algorithms to MIMD medium-grain architectures. First, it is usually possible to concentrate processors at locations where useful work will be accomplished. We allow available processors to perform a step in the algorithm, and simultaneously create a list of locations to be considered in subsequent steps. In much the same way that data-flow techniques analyse the flow of data through processing steps in an algorithm, MIMD architectures in image processing demand a "task flow" analysis. The data can be regarded as fixed, like a blackboard, but the tasks that must be assigned flow in a way which depends upon the initial data. Consider, as a trivial example, an erosion operation on a binary image. Erosion, for our purposes, can be defined as setting all boundary pixels with value "1" to "0", thereby eroding the border or regions. The SIMD approach to this operation is to replace each pixel's value with the minimum in the local neighbourhood. However, in the SIMD approach, we note that "useful" work is accomplished only on the border of regions. The MIMD approach would therefore be to create a list of points on the borders of regions, and to assign processors to the tasks of resetting those pixels to the value "0". At the same time, processors can determine a new list of boundary points, where the next iteration of an erosion operation will create new pixel values. The new list might contain redundancies, but it is nonetheless a considerably smaller list than the set of all pixels. In this algorithm, we see that the tasks are assigned to border pixels, and that the border flows in a predictable way, such that a new list of border points can be constructed from an iteration applied to previous border pixels.

A second concern is one of synchronization. The SIMD algorithms assume that many tasks are accomplished concurrently, and normally assume that concurrent reads are allowed and yield identical information. Frequently, concurrent writes are also permitted, and it is generally assumed that exactly one write to a specific location at a given time is successful. When we convert to an MIMD environment, in image-processing applications we are normally in a processor-poor situation. Thus tasks that were meant to be done concurrently will be at least partly serialized. Standard issues of data concurrency arise: if early serialized subtasks change the values of data, then later tasks may access data that is different from what would be expected in a synchronous application of all tasks. These problems can be handled by one of three methods: (i) read from a fixed copy of the data, and write a new copy with appropriate changes; (ii) read from the data, writing changes as a list of tasks to be performed later, after all of the serialized analysis tasks are finished; or (iii) show that asynchronous operation of the steps, which is what partial serialization with early writes to the data constitutes, is a satisfactory approach for the algorithm in question. For the erosion example, the third alternative is

probably not appropriate, since changing a border pixel from a "1" to a "0" changes what might be considered a border pixel, so that if the list of tasks contains border candidates that are not border points, a subsequent task might change a pixel which otherwise would be considered an interior point. Of course, if the list of points is always known to be a (possibly redundant) list of border points, then the tasks amount simply to setting values to zero, and determining new border points, and asynchronous operation will work. Otherwise, the second approach, of creating a list of border points to be converted to "0"s and to enqueue new candidate boundary points on a new list, will be necessary.

2 A PARALLEL ALGORITHM FOR CONNECTED COMPONENTS OF A GRAPH

Shiloach and Vishkin [11] give an algorithm to find connected components of an undirected graph using a computational model which allows simultaneous reads and writes by multiple processors to a common memory. In the case of simultaneous writes, the model specifies that one of the writes succeeds, but does not specify in advance which processor will be successful.

Each vertex in the graph is assigned a *parent* pointer. The algorithm uses these pointers to construct a forest of root-directed trees, also called a *pointer graph*; each tree is composed of vertices known to reside in the same connected component. Initially each vertex is its own parent, i.e. each vertex comprises a rooted tree with only one node. At termination, the parent pointers of all vertices in the same connected component point to the same root vertex.

In the Shiloach/Vishkin parallel algorithm, each vertex is assigned a processor. Each edge is assigned two processors; in effect, a processor in each direction of edge traversal. Two operations are performed in the algorithm. The *shortcut* operation redirects a vertex's pointer to the parent of its parent, reducing the height of the trees. Shortcutting is mediated by the vertex processors. The *hook* operation directs the pointer of the root of a tree (self-directed before the operation) to point to some node in another tree. Hooking is mediated by the edge processors. We let V be the set of vertices and E the set of directed edges, one for each edge processor.

At intermediate stages of the algorithm, the trees in the pointer graph can be classified as *live*, *stagnant* and *dead*. A live tree has been shortcut or hooked in the current iteration. A stagnant tree has not been shortcut or hooked in the current iteration. A dead tree has not been shortcut or hooked in the previous iteration.

In the algorithm that follows, each vertex is assigned an *age* value, which is

used to determine when trees are stagnant or dead. Further, we assume that each vertex v is denoted by a distinct integer. Thus if v is the integer value for a vertex, parent[v] is the integer value of v's parent vertex, and age[v] is the integer value of the last iteration in which v was shortcut or hooked.

```
        /* initialize */
0)      (∀v)[v ∈ V] cobegin
          parent[v]←v;
          age[v]←0;
        coend
        I←0;
        While (∃v)[age[v] = I] begin
          I + +;
          /* Shortcutting */
1)        (∀v)[v ∈ V]cobegin
            old-parent←parent[v];
            parent[v]←parent[parent[v]];
            if (old-parent≠parent[v]) age [parent[v]]←I;
        coend
        /* Ordered Hooking */
2)      (∀ (x,y))[(x,y) ∈ E] cobegin
          If parent[x] = parent[parent[x]] /* x points to a root */
          and parent[x] < parent[y] then begin
            parent[parent[x]]←parent[y];
            age[parent[y]]←I;
          end
        coend
        /* Stagnant Hooking */
3)      (∀ (x,y))[(x,y) ∈ E] cobegin
          if parent[x] = parent[parent[x]] /* x points to a root */
          and age[parent[x]] < I /* root is stagnant */
          and parent[x] ≠ parent[y] then begin
            parent[parent[x]]←parent[y];
            age[parent[y]]←I;
          end
        coend
        /* Shortcutting */
4)      (∀v)[ ∈ V] cobegin
          old-parent←parent[v];
          parent[v]←parent[parent[v]];
          if (old-parent ≠ parent[v]) age[parent[v]]←I;
        coend
      end
```

3 TIME COMPLEXITY OF THE ALGORITHM

The correctness of the algorithm is easily established. We shall concentrate on the time bound. The proof sketched below differs only slightly from that of [11]. In [11] a bound on the cardinality of the trees is related to the iteration counter I. Here, we show instead a bound on the *height* of the live trees in the forest formed by the parent pointers. We define height as the maximum number of pointers that must be traversed in a simple path from any vertex in the tree to the root, with the additional proviso that tree height is always at least one. In particular, a tree consisting of only a single node (the root node with a self-loop) is considered to have height one, as is a tree where all vertices point to the root.

We first quantify the effect of shortcutting with the following Lemma.

Lemma 1

Shortcutting a tree of height $h > 1$ reduces its height by at least a factor of 3/2.

Proof For any tree of height $h > 1$ shortcutting yields a height of $\lceil h/2 \rceil$. But for any $h \geq 2$ we have

$$\frac{h}{\lceil h/2 \rceil} \geq 3/2.$$

□

We shall also need a rather technical lemma.

Lemma 2

If $u >$ parent$[u]$ after any step in the algorithm then either u is a leaf node or all of u's predecessors in the pointer graph are leaf nodes.

Proof The proof is by induction on the steps in the algorithm. Initially, since parent$[u] = u$ for all vertices, the lemma holds true vacuously. If the lemma holds true at the start of a shortcutting step (Steps 1 or 4), then the shortcutting operation of pointing u to w where there were links from u to v and v to w can create a new link with $u > w$. However, in this case, either $u > v$ or $v > w$ before the shortcutting. In the former case, either u is a leaf or all its predecessors are leaves. In the later case, v's predecessors, including u, must be leaves. In either case, after the shortcutting, u must be a leaf.

Ordered hooking (Step 2) can never create a link from a larger to smaller vertex.

Stagnant node hooking (Step 3) can link a node u to a node v with $u > v$.

However, in this case, u must be a root of a stagnant tree. A stagnant tree cannot have a height greater than 1, or else it would have been shortcut in Step 1. Thus at the start of Step 3, u is either a leaf, or all of its predecessors are leaves. During Step 3 no other stagnant root can hook onto u, since otherwise in Step 2 one of the two roots would have been eligible to hook onto the other. Thus in Step 2 either u would have succeeded in hooking onto something, in which case it is not a root, or something succeeds in hooking onto u, in which case it is not stagnant. Further, during Step 3, any of u's predecessors will remain leaves, since nothing can hook onto a leaf. Thus at the end of Step 3 the lemma holds for node u. This establishes the induction for steps 1, 2, 3 and 4. □

Lemma 3

If n trees of height h_i, $i = 1, \ldots, n$, are hooked together, the result is a tree of height no more than Σh_i.

$$\sum_{i=1}^{n} h_i.$$

Proof The first part of this lemma is to show that the result is a tree; that is, we must show that no cycles are created. Suppose that a cycle is created after some step. Then somewhere along the cycle there must be a pair of consecutive nodes u and v such that $u > v$ and u points to v. According to the previous lemma, either u is a leaf, or all of u's predecessors are leaves. Either way, u cannot be part of a cycle.

Consider hooking T_1 into T_2, assuming both trees have more than one node. Let their heights be h_1 and h_2 respectively. After hooking, the maximum height h of the new tree is no more than the sum of the heights of the old trees, plus one for the new link, less one since we could not hook to a leaf in T_2; thus

$$h \leq h_1 + 1 + h_2 - 1 = h_1 + h_2.$$

If either or both trees has one node, that node can contribute at most 1 to the height of the resulting tree, so the result follows. The result easily generalizes when n trees are hooked together. □

Definition

A root u is *eligible* if there exists an edge $(x,y) \in E$ and a node $v \neq u$ with parent$[x] = u$ and parent$[y] = v$, such that either $v > u$ or age$[u] < I$.

Lemma 4

Eligible roots are always hooked.

Proof Suppose u is an eligible root, and (x,y) is the edge with parent$[x]=u$ and parent$[y]=v$. If $v>u$ then in Step 2 the edge processor (x,y) will attempt to hook u onto v. Either (x,y) succeeds or some other processor succeeds; in either case, u will hook onto some node. If age$[u]<I$ and u does not hook in Step 2 then u is a stagnant root by Step 3, so the processor (x,y) will attempt to hook u to v. Once again, some such processor succeeds, and so u hooks. □

Lemma 5

Dead trees stay dead.

Proof Recall that a dead tree is one that has not been shortcut or hooked in the previous iteration. Clearly the height of a dead tree must be 1, or it would have been shortcut. Also, its root must not be eligible, or it would have been hooked (Lemma 3). Thus it can only be brought back to life if another tree is hooked into it, which implies the existence of some edge (x,y), where x is in the dead tree and y is not. But then the root would be eligible, since u must point the root, which is stagnant by Step 3 of the iteration in which the tree dies, and we have a contradiction. Thus if a tree is dead, it cannot be hooked into or shortcut, so it stays dead. □

We can now present the key

Time bound

The sum of the heights H_i of all live trees at the end of iteration i is bounded by

$$H_i \leq \left(\frac{2}{3}\right)^i N,$$

where N is the number of vertices in the graph.

Proof By induction on i. Initially $(i=0)$, each vertex forms its own tree,

of height 1, so $H_0 = N$. The induction hypothesis states that after $i-1$ iterations the sum of the heights H_{i-1} of the live trees is bounded by

$$H_i - 1 \leq \left(\frac{2}{3}\right)^{i-1} N.$$

We show that a tree in the forest of height h at the end of iteration $i-1$ contributes no more than $\frac{2}{3}h$ to H_i. If $h=1$ and the root is not eligible, the tree dies, and it contributes nothing to H_i. If $h>1$ and the tree is neither eligible nor hooked into by some other tree, it is shortcut in Step 4, and its contribution to H_i is at most $\frac{2}{3}h$. If the root hooks or is hooked, then the height of the new tree to which it belongs is no more than the sum of the old heights of the individual trees which have hooked together. However, since the new tree has height greater than 1, and Step 4 will shortcut the tree, its height will be at most $\frac{2}{3}$ the height of the sum at the end of the iteration. Thus each participating tree contributes no more than $\frac{2}{3}$ its height in the previous iteration to H_i. □

As a direct consequence of the bound above, we see that after more than $\log_{3/2} N$ iterations, we must have $H_i = 0$, which is to say that all trees are dead, and the algorithm terminates.

Note that the Step 1 shortcutting is necessary to define eligibility of a node (a stagnant node). Since Step 1 may be applied to trees of height 1 which do not reduce in height and yet do not die because they can hook, we cannot claim a 3/2 factor total height reduction on account of Step 1.

4 NECESSITY OF THE STEPS

As noted in [11], Step 4 is unnecessary, and omitting it will at worst double the number of iterations required for termination. Its inclusion simplifies the proof, and less than doubles the amount of work required per iteration while halving the number of iterations that will be needed in the worst case. Whether the trade-off is reasonable in the average case is best judged by empirical studies.

Step 2 is needed to prevent cycles from forming. If Step 3 is omitted then the algorithm will still work well—however, the $O(\log N)$ time bound is no longer valid. Dropping Step 3 is appealing, however, since it is the only step that introduces pointers to smaller vertices. Without Step 3, the final labelling will be components with vertices pointing to the maximum vertex within each component. If the vertices are numbered according to some labelled value, this provides an algorithm for simultaneously labelling connected components and finding maximum values within components. If

the $O(\log N)$ connected components algorithm is performed first, and then maxima found within each component, the result for an Ultracomputer is an $O(C \log N)$ method, where there are C components. If the Ultracomputer permits parallel maxima, in the same way that it permits parallel adds, then the method is $O(C + \log N)$. Thus if omitting Step 3 retains the $O(\log N)$ performance of the algorithm, we have a better component maximum method; although the advantage to the case where parallel maxima are allowed is very slight.

Unfortunately, as is shown in [11], the algorithm without Step 3 can result in $O(N)$ time performance. The situation arises with a *star* graph. We might conjecture that for subgraphs of images, which are planar graphs with bounded in-degree, such examples will not arise. However, Fig. 1 shows a *comb* graph, which when numbered as shown, and under somewhat perverse conditions, will require $O(N)$ iterations. The first hook operation can yield the pointer graph shown in Fig. 2. The only eligible root is $n+1$, which in the next iteration might hook to $n+2$. Continuing in this way, a possible sequence of $n/2$ stagnant node hooks will be needed.

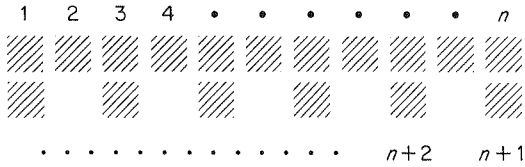

Fig. 1. A comb graph.

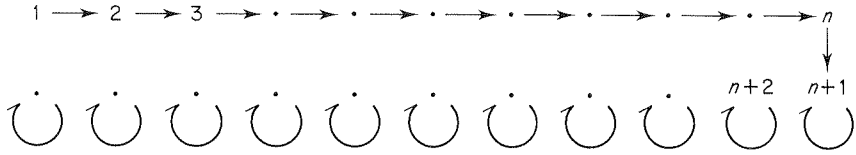

Fig. 2. The pointer graph after the first min-neighbour hook operation. If stagnant node hooking is omitted, then after a couple of short-cut operations, the only eligible root is "n+1", which might then hook to "n+2". This process can continue, yielding n/2 iterations before termination.

If the graph is ordered in raster-scan fashion, instead of with malicious intent as was done in the example above, it is still possible to construct an example which needs linear time. In this case, however, the comb has to be laid out in one dimension across the image, and so an n by n image is limited to $O(n)$ complexity.

The perverse nature of these counterexamples leads us to conjecture that an expected time of $O(\log N)$ would be obtained by the simplified algorithm omitting Step 3 if all hooks are equally likely.

5 TASK FLOW OF THE ALGORITHM

The proof of the algorithm allows us to make some immediate observations about when processors will be needed for useful work.

Observation 1

All vertices in a dead tree and all edges adjoining vertices in a dead tree will be inactive for the remainder of the procedure.

This follows immediately for Lemma 5. □

Observation 2

Once it is known that two neighbouring vertices x and y point to nodes in the same tree, then the edge processor (x,y) will no longer be needed.

This follows because edge processors are only used for hooking roots to other nodes, and since cycles are never created (Lemma 3), roots never hook to nodes within the same tree. □

As a corollary to Observation 2 we have the following.

Observation 3

If parent[x] = parent[y] then (x,y) will thereafter be inactive.

Observation 4

If (x,y) succeeds in hooking the root parent[x] onto the vertex parent[y] in either Step 2 or 3 then the processor (x,y) will be inactive thereafter.

Given the inclusion of Step 4, we can identify yet another situation when processors will be inactive:

Observation 5

If, during Step 4, parent[u] is set to point to a root then the vertex u need not be considered in the subsequent Step 1.

Of course, shortcutting is unnecessary if u points to a root. However, it takes as much time to check whether u points to a root as it does to execute the shortcutting step for u. Thus this observation would not seem to help much. But if a vertex points to a root at the start of Step 4, then it will be among those vertices that point to roots after Step 4, and this information can be obtained essentially for free (i.e., parent[u] does not change in Step 4). These nodes need not be considered in the following Step 1.

Note that, owing to the possibility of hooking in Steps 2 and 3, all vertices except those in dead trees must be subjected to shortcutting in Step 4.

We see that for the most part, tasks to be performed by steps in the algorithm are pruned as processors become inactive. The only exception is that shortcutting a node in Step 1 can be omitted if the node is known to point to a root on that iteration; it may happen that the root will subsequently be hooked and so the processor will be reactivated. Nonetheless, we might be tempted to maintain a list of active processors, and remove items from the list as processors die according to the observations above. However, for purposes of synchronization, we need to form a new list for each step, and to read from the current list when assigning tasks in the current step. Thus it is easier to enqueue tasks that will be necessary in subsequent steps as they are discovered in the current step. We discuss the enqueuing and processor assignment methods in the implementation section.

How should task requests be represented when enqueued on a list? For Steps 1 and 4, tasks will be represented as a list of vertices to be subjected to a shortcutting step. For Steps 2 and 3, tasks are given as a list of edges between vertices. Strictly speaking, the edges are directed. However, if (x,y) is inactive, then (y,x) is also inactive, since our only criterion for inactivating edge processors is Observation 2, which is symmetric in x and y. So the entry (x,y) can stand for two edge tasks: namely (x,y) and (y,x). Further, Steps 2 and 3 can be recoded to perform both of the directed tasks at once as a single task. Initially, all vertices in all components are on the vertex list, and the edge list consists of all edges (x,y) with y as a south or east neighbour of x, and x and y both are vertices in the vertex list. The initial lists can be formed by a raster scan of the image, which can be done in parallel. (See the

pseudocode for Step 0 in the Appendix.) Processors are assigned to successive pixels in raster-scan order. A processor assigned to pixel x enqueues the south and/or east edges (x,y) if the edge connects vertices in regions.

Step 1 is performed for all vertices on the vertex list. No new lists are formed during Step 1.

For Step 2 we examine all edges on the edge list. During Step 2, (x,y) attempts to hook parent[x] onto parent[y] or vice versa, under the conditions that one node is a root and the other has a larger numerical value. A new edge list is formed, and (x,y) is enqueued on that list unless (1) it is observed that parent[x] = parent[y] when checking the order relationship on the parents, or (2) the task (x,y) is successful in hooking parent[x] onto parent[y], or (3) the task (y,x), which is performed as part of the (x,y) request, is successful in hooking parent[y] onto parent[x]. It appears as though a slight bit of additional work is needed to perform these checks: ordinarily, parent[y] does not have to be accessed by (x,y) if parent[x] is not a root. The objection is moot, however, since the symmetric (y,x) operation must check parent[y] to see if it is a root, so that both parents will be accessed anyway. There is, nonetheless, a slight time penalty to determine whether a request to hook is successful, and also to perform the enqueue step, as we will discuss in the implementation section.

Step 3 then uses the new edge list, and can form yet another edge list. If a new list is formed, the process is identical with the edge list formation in Step 2: for each edge (x,y), parent[x] and parent[y] will both be accessed (to see if either is a stagnant root). The edge (x,y) is not enqueued if parent[x] = parent[y], or if parent[x] succeeds in hooking onto parent[y], or vice versa.

It is not clear that forming a new list (which will be used for Step 2 in the next iteration) is worth the extra expense during Step 3. Instead, one can simply retain the edge list formed during Step 2, and use it for the next iteration's Step 2. Some edges will not be pruned that are otherwise unnecessary, but they are likely to be rare, and will be pruned during the list formation in the subsequent Step 2. Empirical studies are needed to judge whether creating a new list in Step 3 is justified.

Finally, Step 4 uses the vertex list, and creates two new vertex lists. One vertex list will be used for the subsequent Step 1, and the other vertex list will be held for Step 4 of the next iteration. The Step 1 list is formed of all vertices that change parents during the Step 4 shortcutting. The remaining vertices are known to point to roots, and so need not be considered in the subsequent Step 1. The Step 4 list is formed of all vertices except those in dead trees. A member of a dead tree can be recognized in Step 4 by noting whether the vertex points to a root which has not been shortcut or hooked in

Steps 1, 2, or 3. That is, the vertex v is enqueued on the next Step 4 list unless parent[v] does not change and age[parent[v]] $< I$ in Step 4.

Pseudocode for the modified steps is presented in the Appendix.

6 SYNCHRONIZATION

In a processor-poor environment we cannot execute all of the necessary tasks within a step at once. Since we wish to avoid copying the entire pointer graph and age labels to a separate output graph in each step, we must either queue the requests for changes to the pointer graph, and execute the changes after completing the analysis for the step, or suffer the consequences of asynchronous operation of each step. Since we have a list of tasks to be performed, we could enqueue the results of those tasks to a separate list, being sure to request a hook on a root no more than once (the first request to a node should block all further requests), and storing with all shortcut requests the node to which the new parent should point. The request list could be performed after the task list is exhausted (synchronizing between scans on the lists). It turns out, however, that in the case of the Shiloach/Vishkin connected-components algorithm, it is sufficient to operate each step asynchronously.

Within Steps 1 and 4, if shortcut requests operate asynchronously, trees will still reduce in height. The final tree may be different than the tree resulting from a synchronous shortcut step. For example, a four-node linear tree of height 3, if shortcut by 4 separate (sequential) tasks, starting from the root and propagating to the leaf, will result in a rooted star. The same tree, when shortcut by a parallel operation, results in a tree of height 2. However, in this example and all other examples, asynchronous shortcutting can only lead to more height reduction as compared to synchronous shortcutting. This is because a node's parent's pointer may have already been shortcut, in which case it can only point closer to the root. Thus the shortcut on the current node will yield a parent pointer that goes close to the root. Unfortunately, an improvement cannot be guaranteed, since it may happen shortcut tasks occur from the leaf nodes toward the roots, rather than the other way around.

If hooking operates asynchronously, the first time a root u is hooked to a node v, it ceases to be a root, and so will receive no more hook requests during that step (or, for that matter, in subsequent steps). However, the new edge from u to v might be responsible for causing another node u' to attempt to hook to v during the same step. Alternatively, if the node v happens to be a root, it might happen that it will attempt to hook onto a node u' on account of the new edge from u to v during that step. In either case, the situation arises

if u has some neighbour x such that x points to u'. However, an analysis of the proof of the time bound shows that the only requirement of Steps 2 and 3 is that all eligible roots going in to a hooking step are in fact hooked. Asynchronous operation does not defeat this property, since asynchronous hooks only add possibilities for new hooks. None of the new hooks are harmful, since they only connect trees that comprise nodes in the same connected component. Some of the hooks may speed up the algorithm, by combining trees that otherwise would not be combined until later iterations.

We conclude that asynchronous parallel processing of the steps will at worst speed up the algorithm in terms of numbers of iterations.

The algorithm as given requires synchronization between the steps. Synchronization between Steps 2 and 3 is essential to properly define stagnant roots, and to prevent cycles. Synchronization between Steps 3 and 4 is needed to define dead trees, and to ensure that Step 4 does the useful work of reducing the height of combined trees. That is, if some shortcutting steps of Step 4 were to precede stagnant node hooking in Step 3, shortcutting might be attempted on a stagnant tree which is later hooked, and thus the combined tree height is not reduced by the requisite factor of 3/2. Of course, the tree height would be reduced in the next Step 1, and so eliminating synchronization between these two steps is at worst equivalent to eliminating Step 4 entirely, which simply slows the algorithm by a multiplicative factor. Between Steps 4 and 1, synchronization is needed to properly test for termination, and again to ensure that Step 4 accomplishes all the work credited to it. The iteration counter is increased in a serial section between these two steps, and the test for termination is also done by a single serial processor. However, we could make the iteration counter a local variable, updated by each processor, and observe vertices not pointing to roots in Step 4, feeding them into processors which then execute the Step 1 shortcutting. This of course assumes a partly asynchronous operation of the steps, and means that a Step 1 shortcut on a node v will access v's parent, who may not yet have been shortcut by its Step 4 operation. The result is that Step 4 may be partly ineffectual, which as noted before at worst slows down the algorithm by a constant factor.

There is a minor penalty involved in requiring synchronization. Especially if the number of processors P is a large fraction of the number of vertices N, there will be a point at which all tasks for a step have been assigned and many completed, and available processors have nothing better to do than wait for completion of the remaining uncompleted tasks. Thus the degree of parallelism is hurt by the need to keep some processors idle.

7 IMPLEMENTATION

The Appendix contains pseudocode for performing the Shiloach/Vishkin algorithm on an MIMD machine. We have written the program assuming synchronization between all steps, although as noted in the previous section, synchronization is necessary only between Steps 2 and 3. We also assume asynchronous operation within each step.

In our C-like language, variables declared as "shared" are stored in the common memory, and so are accessible to all processors. The address of a shared variable (denoted by &v for the variable v) is also available to all processors. We have avoided declaring automatic or static shared pointers for clarity's sake; we could otherwise declare shared pointers (or variables in common memory containing addresses) pointing to shared variables, or local copies of addresses pointing to shared variables, or even perhaps pointers in common memory pointing to local variables, and it is not so obvious how the syntax should differentiate between these cases. In our algorithm, some of the procedure parameters are pointers to shared integers, and it is understood that the addresses of those shared integers are available to all processors within that procedure. That is, in

```
sub(pN)
shared int *pN;
{ . . .
}
```

all processors operating during { . . } have access to the integer *pN, and a copy of the address pN will also be located in common memory, but may well be cached by individual processors in their local memory if the processors handle the pointer as a read-only variable.

A block-structure language such as ADA or PASCAL might be more appropriate for coding these kinds of parallel algorithms, since the scoping rules for nested blocks would make the visibility of variables explicit.

We have used the "fetch-and-add" instruction as our principal coordination primitive. This instruction accesses a shared variable, returns the value, and increments the stored amount by a specified increment. That is, the instruction is equivalent to

```
fetch&add(pV, inc)
int *pV;
int inc;
{
    int val
    val = *pV
    *pV = val + inc
    return(val)
}
```

The important point is that concurrent fetch-and-adds to the same shared variable behave as though they had been requested serially, in some arbitrary order. Thus ten simultaneous "fetch&add(&v,1)" requests will increment v by 10, and return ten distinct consecutive integers, one to each calling processor. An important property of the Ultracomputer is that simultaneous fetch-and-adds to a single variable incur no time penalty. That is, all processors can issue a fetch-and-add to the same variable in the same amount of time as it would take for one processor to execute a fetch-and-add.

We use the construct "cobegin . . . coend" to allocate processors. The "cobegin nomorethan N"

line denotes a request to the operating system to assign as many processors as are available, but no more than the integer value in the variable N, and put those processors to work executing the following code. The processors use the "fetch-and-add" primitive to dynamically allocate unique identifiers, which leads to data variability and hence execution variability amongst the processors. The multiple processors thus execute the intervening code in an asynchronous fashion. The "coend" line is interpreted as requesting that all but the last assigned processor reaching the "coend" statement be yielded back to the operating system, and perhaps put into an idle loop. The last processor to reach the "coend" statement proceeds to execute the code following the "coend".

The operating system might implement the "cobegin . . . coend" construct by spawning exactly the requested number of processes upon reaching the cobegin, leaving it to the scheduler to share processors among requested processes. A shared variable is loaded with the number of active processes. As a process completes, by reaching the "coend" statement, a "fetch-and-add" instruction with an increment of -1 is issued to the shared variable representing the number of active processes. The last process will receive the value 1 from the fetch-and-add instruction, and thus will know not to die, but instead to continue by reading instructions following the "coend".

An alternative approach allows the "cobegin" to instantiate available processors to the intervening code, along with a shared variable which counts the number of assigned processors. As processors reach the "coend", they issue fetch-and-adds with an increment of -1. Once again, the last processor will receive a value of 1, and thus know to continue with the remaining code.

Within each "cobegin . . . coend" block, we consider each step as a set of independent and potentially parallel subtasks. The subtasks are given by the processing of each vertex or edge. We organize the vertices and edges into lists, which are represented by arrays indexed by a shared variable. In the

fully parallel case, if a sufficient number of processors are available, we use a processor for each vertex or edge element on the list. A processor requests a new subtask by executing a fetch-and-add to the shared index with an increment of 1. The value returned is a pointer to the element (edge or vertex) that is to be processed in the subtask. The nature of the fetch-and-add instruction ensures that no two processors will receive the same element. The guarding "while" loop ensures that all tasks are assigned before any processor is yielded to the "coend" statement, and that any available processor requests the next unassigned task.

Note that if the allocation of processors is done with a fixed unique process identification number for each, then that number can be used to coordinate the allocation of tasks so that no two processors are assigned the same task. However, the dynamic allocation of tasks becomes much more difficult without a fetch-and-add instruction. The time penalty for synchronization between steps is likely to be much greater if the task allocation is based on static processor identification numbers.

In our code for the Shiloach/Vishkin connected-component algorithm, we assume that the image is a globally available raster-scan array of binary values (1s and 0s). We construct the pointer graph, which will also be global. We structure the pointer graph as an image of indices in registration with the input image: at a given pixel, the value of "Parent(i)" at that pixel is the index of the pixel pointed to by that node in the pointer graph. This structure makes the code more clear, but wastes a lot of space, since Parent(i) is undefined for pixels i that are not within regions in the binary image (i.e. where Img(i) = 0). Instead, the pointer graph should be a set of nodes, with each node containing fields specifying the coordinates of the corresponding pixel, the Age field, and a pointer to the parent node in the pointer graph. Since our pointer graph is in registration with the image, and indexed by the pixel coordinate, the Age field is also a registered image. We also use a "Hooked" tag on each node, which is used to determine whether a node is successful in hooking onto another node in Steps 2 and 3.

The vertex and edge lists are simply arrays of indices and pairs of indices respectively. We use the fetch-and-add primitive to enqueue elements onto the lists. Initially, all vertices within regions in the image are placed on the vertex list, and all south and east edges in regions are placed on the edge lists, as built up in Step 0. Step 1 shortcutting is unnecessary during the first iteration, and so is skipped. In all of the steps, a fetch-and-add is issued to an index in order to allocate elements on the list to processors. In Steps 2 and 3 the current edge list is scanned, and a new edge list is formed. Step 2 creates Step 3's edge list, while Step 3 creates the edge list for the next Step 2. Each processor enqueues the edge element it is currently processing onto the output list, unless it observes that both vertices of the edge point to the same

node, or unless the current edge is successful in causing a hook. Since several processors may request a hook of a single node simultaneously, it is necessary for a processor to determine whether it has been successful in causing a hook. This is done by means of a hook count attached to each node, represented as the array "Hooked(i)". In order for a processor to gain permission to hook node i, it must issue a fetch-and-add with increment 1 to the variable Hooked(i), and retrieve the value 0. Since only one processor will receive a 0-value from the fetch-and-add, exactly one hook will take place and a processor knows whether it is the winner according to the permission granted from the returned value. In this scheme, we are using the fact that once a node hooks onto another node, then it is no longer a root and will at no future time be eligible for hooking again.

During Step 0 the vertex list for Step 4 is created, and stored in the array Vlist4a. The vertex list for Step 1 is not needed, since Step 1 is skipped during the first iteration. During Step4 the Vlist4a list is read, and new lists of vertices are created, one for Step 1 and one for the next iteration's Step 4. The new list for Step 4 is written to Vlist4b. If we instead wrote to Vlist4a, overwriting the list being scanned, we could potentially overwrite a value in the list that has been allocated to a processor but not yet read. The length of the list, NV4, is continually overwritten, but the length of the original list is passed by value and stored in the procedure-local variable NV. In alternate iterations we then read from Vlist4b and write to Vlist4a. Finally, note that Age fields are updated after each shortcut and hook operation, updating the target of the shortcut or hook in Steps 1, 2 and 3. A dead tree is one whose root is stagnant after Step 3. Note that a hook into a stagnant tree must hook onto the root, so that if a stagnant tree is made live by Steps 2 or 3 the Age field of the root is updated. Thus in Step 4, nodes in dead trees are recognized because they point to roots whose Age field has not been updated in the current iteration.

Our time bound of Section 3 shows that the algorithm will terminate after no more than $\log_{3/2} N$, i.e. approximately 1.71 $\log_2 N$ iterations. Inspecting the algorithm, we see that Step 1 requires 5 accesses to the common memory, while Steps 2, 3 and 4 require 11 accesses per task. The time used for Step 0 is negligible. In counting the number of accesses, we have assumed that a fetch-and-add is equivalent to two accesses, that independent reads or writes can be pipelined through a connection network and thus can be performed essentially concurrently, and that accesses by separate processors do not interfere. These are all reasonable assumptions, modulo minor perturbations, for the Ultracomputer.

The unknown variable in an analysis of the MIMD approach to the Shiloach/Vishkin algorithm is the rate of decrease of the task list sizes. Initially, the number of tasks are $NV1 = NV4 = N$ and $NE2, NE3 < 2N$. We

conjecture that the edge lists will normally drop off geometrically. The vertex lists, particularly the list for Step 4, will drop only after a component is completely labelled, which can take 1.7 log N_c iterations, where N_c is the size of the component. In any case, no iteration will need more than $60N$ accesses, and considerable drop-off is expected as iterations proceed.

Let N_c be the number of nodes in the maximum size connected component and N the total number of nodes. Suppose that there are P processors, and each access to shared memory takes $\alpha \log P$ time (in seconds). Since P processors can be devoted to performing tasks simultaneously, we come up with a time bound of

$$\frac{103\alpha N \log N_c}{P/\log P}$$

seconds for termination of the algorithm. If we dispense entirely with the enqueuing of new tasks, using the same vertex and edge lists in all iterations, we can then modify the code so that 5 accesses per task are needed in Step 1, and 6 accesses per task suffice in Steps 2, 3 and 4. In this case, we know that roughly 35 N accesses will be needed per iteration, with no drop-off. We can thus drop the 103 constant in our upper bound to 60, for otherwise the enqueuing process is not supportable. The expected constant is probably somewhat smaller, due to the drop in the length of the task lists.

We can also inspect the code to determine what percentage is spent in allocation and enqueuing of tasks, and what percentage is spent in actually performing the steps of the algorithm. This gives a measure of the degree of parallelization. The measure is still somewhat misleading, however, since the use of an interconnection network or other access strategy exists solely to support concurrent reads and writes, and is thus an additional price paid for parallelism. The synchronization between steps introduces an additional drop in the degree of parallelization, which is difficult to measure since it depends upon the load on the parallel computer and the number of available processors relative to the size of the task lists. Finally, many of the tasks attempt to perform a shortcut or hook and are unsuccessful in the attempt, either because of multiple requests or because the conditions are not satisfied to request the action. These tasks might be considered wasted effort. Just taking into account allocation and enqueuing, we estimate that nearly 50% of the accesses during shortcutting steps and only 30% during hooking steps constitute performance of the algorithm. The rest is overhead for parallelization.

Typical numbers for a 512 by 512 image would be $\alpha = 25$ ns, $N = 512^2/8$, $N_c = 4000$ and $P = 4096$. This yields a time bound (using the constant 60) of 1.728 ms.

This compares with the standard sequential methods, which permit two

frame-time (67 ms) connected component labellings on 512 by 512 images. By using a supercomputer with a 10 ns cycle time, a linear connected component can be performed in something like $10n^2$ cycles, which for $n = 512$ yields 26 ms times.

Let us briefly compare these times with an n by n mesh connected parallel array of single-bit processors. Let us assume the breadth-first-search method instead of the Nassimi/Sahni algorithm. Each pixel requires a unique processor identifier, which for an n by n mesh will have $2 \log n$ bits. A local max among the central pixel and the four nearest neighbours can be constructed using a sequence of 5 pairwise maxima. The maximum of two integers a and b can be calculated using the formula $\max(a,b) = \frac{1}{2}(a+b+|a-b|)$. Thus roughly 10 arithmetic operations, each involving integers with $2 \log n$ bits, are required per iteration. In addition, a $2 \log n$ bit operation is needed to mask the boundary, and another operation is needed on each iteration to check for termination. We assume the existence of a global "sum-or" which can detect a non-zero value within a designated plane in the grid in one cycle time. Combining, we estimate that $24 \log n + 1$ cycles are needed per iteration. How many iterations will be needed, on the average? We are tempted to answer $N^{\frac{1}{2}}$ as an estimate of the diameter of the largest component. Unfortunately, the components are frequently boundaries of connected regions, and a safer bet for the maximum internal path length is $\frac{1}{2}n$. Thus we arrive at the time bound of

$$\alpha n\,(12 \log n + \tfrac{1}{2}).$$

Using $n = 512$ and $\alpha = 50$ ns, which seem to be currently typical numbers, we obtain a time bound of 2.78 ms.

We see that for these numbers, the MIMD Shiloach/Vishkin approach wins, although the bounds are within the same order of magnitude. Thus the true benefits of MIMD approaches to image processing lie in the greater degree of flexibility, and performance improvements, if present, will be borne out mostly by realistic empirical studies.

ACKNOWLEDGMENTS

Work on this paper has been supported in part by NSF Grant DCR-8403300. An earlier draft and much of the research was conducted by Alan Rojer, whose assistance is greatly appreciated. He was supported by Navy Grant N00014-85-M-0260.

REFERENCES

[1] Duff, M. J. B. (1978). Review of the CLIP image processing system. In *Proc. Nat. Computer Conf.* p. 1055. See also Duff, M. J. B. (1985). Real applications on CLIP4. In *Integrated Technology for Parallel Image Processing* (ed. S. Levialdi), 153–165. Academic Press, London.

[2] Batcher, K. E. (1980) Design of a massively parallel processor. *IEEE Trans. Comp.* **29,** 836–840.

[3] Kung, H.T. (1982). Why systolic architectures? *Computer* **15,** 37–46.

[4] Hummel, R. A. (1984). Image processing on the NYU Ultracomputer. *Courant Inst. NYU Ultracomputer Note* No. 72.

[5] Nassimi, D. and Sahni, S. (1980). Finding connected components and connected ones on a mesh-connected parallel computer. *SIAM J. Comp.* **9,** 744–757.

[6] Miller, R. and Stout, Q.F. (1984). The pyramid computer for image processing. *Proc. 7th Int. Conf. on Pattern Recognition.* pp. 240–242.

[7] Gajski, D., Kuck, D., Lawrie, D. and Sameh, A. (1983). CEDAR—a large scale multiprocessor. In *Proc. Int. Conf. on Parallel Processing*, p. 524.

[8] Gottlieb, A., Grishman, R., Kruskal, C.P., McAuliffe, K.P., Rudolph, L., and Snir, M. (1983). The NYU Ultracomputer—designing an MIMD shared memory parallel computer. *IEEE Trans. Comp.* **32,** 175–189.

[9] Siegel, H. J., Siegel, L. J., Kemmerer, F. C., Mueller, P. T., Smalley, H. E. and Smith, S. D. (1981). PASM: a partitionable SIMD/MIMD system for image processing and pattern recognition. *IEEE Trans. Comp.* **30,** 934–947.

[10] Reeves, A. P. (1985). Multicluster: an MIMD system for computer vision. In *Integrated Technology for Parallel Image Processing* (ed. S. Levialdi), pp. 39–56. Academic Press, London.

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

APPENDIX. MIMD SHILOACH/VISHKIN CONNECTED COMPONENTS PSEUDOCODE

```
/*  Shiloach/Vishkin algorithm, coded in pseudo-C code for
*   the NYU Ultracomputer
*/

    #define   n=512          /* Image is n by n */
    typedef   pnode=integer range[0..n^2-1];
    typedef   edge=record
              {x:  pnode;
               y:  pnode;}
    /*globals:*/
    shared    binary   Img(0..n^2-1); /* Input image is n by n, raster scan order */
    shared    pnode    Parent(0..n^2-1);
    shared    int      Age(1...n^2), Hooked(1..n^2);
```

```
        procedure Shiloach Vishkin()
        {
        shared    pnode   Vlist1(0..n^2 - 1), Vlist4(0..n^2 - 1);   /* Vertex lists */
        shared    edge    Elist2(1..2n^2), Elist3(1..2n^2);                /* Edge lists */
        shared    int     NV1, NV4, NE2, NE3;  /* Integers giving the length of lists */
        shared    int     I;                   /* Iteration counter */

        /* begin */

  step0(Vlist4a,&NV4,Elist2,&NE2);      /* Create vertex list for step 4 and Edge list for step 2 */
  I←0;
  while (NV4 > 0)                              /* While there are non-dead trees */
    {I←I + 1;
      if (I > 1)step1(Vlist1,NV1);             /* Uses Vertex list Vlist1 */
      step2(Elist2,NE2, Elist3,&NE3);          /* Using Elist2, create Elist3 */
      step3(Elist3,NE3, Elist4,&NE2);          /* Using Elist3, create Elist2 */

    if (isodd(I))
        {step4(Vlist4a,NV4,Vlist1,&NV1,Vlist4b,&NV4);}
    else
        {step4(Vlist4b,NV4,Vlist1,&NV1,Vlist4a,&NV4);}
    }
  }


  step0(Vlist,pNV,Elist,pNE)  /* Initializes pointer graph and loads Vertex list and Edge
list */
  shared      pnode     Vlist(0..n^2 - 1);
  shared      edge      Elist(1..2n^2);
  shared      int   *pNV, *pNE;
  {
  shared      pnode     pI;

  pI←0; *pNV←0; *pNE←0;
  cobegin nomorethan n^2 - n
      private     pnode     i;
      private     int   k;
      {while ((i←fetch&add- (&pI,1)) < n^2 - n)

                                                  /* We assume that the bottom row and
                                                   * right column of Img are all 0's */
                        {if (Img(i) = 1)
                          {Parent(i)←i;           /*Self pointing root */
                          Age(i)←0;
                          Hooked(i)←0;            /* Counter for hook requests */
                          k←fetch&add(pNV,1);           /* Enqueue node i */
                          Vlist(k)←i;
                          if (Img(i + 1) = 1)           /* Enqueue east neighbor */
                              {k←fetch&add(pNE,1);
                              Elist(k).x←i;
                              Elist(k).y←i + 1;}
```

```
                    if (Img(i+n)=1)                    /* Enqueue south neighbor */
                        {k←fetch&add(pNE,1);
                        Elist(k).x←i;
                        Elist(k).y←i+n;}
                    }
                }
            }
        coend
        }

        step1(Vlist,NV)          /* Shortcuts all nodes on Vlist */
        shared pnode     Vlist(0..n^2-1);
        shared int       NV;
        {
        shared int       index;
        index←0;
        cobegin nomorethan NV
            private pnode      old-parent, new-parent;
            private int   k;
            {while ((k←fetch&add(index,1))<NV)
                {old-parent←Parent(k);
                new-parent←Parent(old-parent);
                Parent(k)←new-parent;
                if (old-parent≠new parent) Age(new-parent)←I;
                }
            }
        coend
        }


        step2(Elist,NE,Elistout,pNEout)
        shared   edge    Elist(1..2n^2);
        shared   int     NE;
        shared   edge    Elistout(1..2n^2);
        shared   int     *pNEout;
        {
        shared   int     index;

        index←0
        *pNEout←0;
        cobegin nomorethan NE
            private     edge        e;
            private     pnode       u,v;
            private     int   k,h;
            {while ((k←fetch&add(index,1))<NE)
                {h←1;
                e←Elist(k);                          /* Edge is (e.x,e.y) */
                u←Parent(e.x);
                v←Parent(e.y);
                if (u<v&&Parent(u)=u)
                    {h←fetch&add(&Hooked(u),1);
```

```
                if (h = 0)                  /* Allow hook only if not hooked yet */
                    {Parent(u)←v;
                     Age(v)←I;
                    }
                }
            else if (u > v && Parent(v) = v)
                {h←fetch&add(&Hooked(v),1);
                 if (h = 0)                 /* Allow hook only if not hooked yet */
                    {Parent(v)←u;
                     Age(u)←I;
                    }
                }
            if (u ≠ v && h ≠ 0)
                {k←fetch&add(pNEout,1);
                 Elistout(k)←e;
                }
            }
        }
    coend
}

step3(Elist,NE,Elistout,pNEout)
shared  edge   Elist(1..2n²);
shared  int    NE;
shared  edge   Elistout(1..2n²);
shared  int    *pNEout;
{
shared  int    index;

index←0;
*pNEout←0;
cobegin nomorethan NE
    private    edge      e;
    private    pnode     u,v;
    private    int  k,h;

    {while ((k←fetch&add(index,1)) < NE)
        {h←1;                                   /* Hooked only if h = 0 */
         e←Elist(k);
         u←Parent(e.x);
         v←Parent(e.y);
         if (u ≠ v)
            {if (Age(u) < I && Parent(u) = u)
                {h←fetch&add(&Hooked(u),l);
                 if (h = 0)                 /* Allow hook only if not hooked yet */
                    {Parent(u)←v; /    * Hook u to v */
                     Age(v)←I;}
                }
            else if (Age(v) < I && Parent(v) = v)
                {h←fetch&add(&Hooked(v),1);
                 if (h = 0)                 /* Allow hook only if not hooked yet */
```

```
                        {Parent(v)←u; /* Hook v to u */
                        Age(u)←I;}
                    }
                if (h≠0)
                    {k←fetch&add(pNEout,l);
                    Elistout(k)←e;
                    }
                }
            }
        }
    coend
}

step4(Vlist,NV,Vlist1,pNV1,Vlist4,pNV4)
shared    pnode    Vlist(0..n^2 − 1),Vlist1(0..n^2 − 1),Vlist4(0..n^2 − 1);
shared    int      NV;
shared    int      *pNV1,*pNV4;
{
shared    int          index;
#define   NOT          !

index←0;
pNV1←0; *pNV4←0;
cobegin nomorethan NV
    private       pnode     old-parent, new-parent;
    private       int   k,j;

    {while ((k←fetch&add(index,1)) < NV)
        {old_parent←Parent(k);
         new_parent←Parent(old_parent);
         Parent(k)←new_parent;
         if (old_parent≠new_parent)
             {j←fetch&add(pNV1,1);
              Vlist1(j)←k;}
           if NOT(old_parent = new_parent && Age(new_parent) < I)
             {j←fetch&add(pNV4,1);
              Vlist4(j)←k;}
        }
    }
coend
}
```