INFORMATION AND COMPUTATION 92, 1–47 (1991)

Approximate Parallel Scheduling. II. Applications to Logarithmic-Time Optimal Parallel Graph Algorithms

RICHARD COLE*

Courant Institute, New York University,
251 Mercer St., New York, New York 10012

AND

UZI VISHKIN†

The University of Maryland Institute for Advanced Computer Studies (UMIACS),
College Park, Maryland 20742, and Tel Aviv University,
Ramat Aviv, Tel Aviv 69978, Israel

Part I of this paper presented a novel technique for approximate parallel scheduling and a new logarithmic time optimal parallel algorithm for the list ranking problem. In this part, we give a new logarithmic time parallel (PRAM) algorithm for computing the connected components of undirected graphs which uses this scheduling technique. The connectivity algorithm is optimal unless $m = o(n \log^* n)$ in graphs of n vertices and m edges. ($\log^{(k)}$ denotes the kth iterate of the log function and $\log^* n$ denotes the least i such that $\log^{(i)} n \leqslant 2$). Using known results, this new algorithm implies logarithmic time optimal parallel algorithms for a number of other graph problems, including biconnectivity, Euler tours, strong orientation, and *st*-numbering. Another contribution of the present paper is a parallel union/find algorithm.

1. INTRODUCTION

The models of parallel computation used in this paper are all members of the parallel random access machine (PRAM) family. A PRAM employs p synchronous processors all having access to a common memory.

* This research was supported in part by NSF Grants DCR-84-01633, CCR-8702271, CCR-8902221 and CCR-8906949 by ONR Grant N00014-85-K-0046, by an IBM faculty development award, and by a John Simon Guggenheim Memorial Foundation Fellowship.

† This research was supported in part by NSF Grants NSF-CCR-8615337, DCR-8413359 and CCR-8906949, ONR Grant N00014-85-K-0046, by the Applied Mathematical Science subprogram of the office of Energy Research, U.S. Department of Energy under Contract DE-AC02-76ER03077, and by the Foundation for Research in Electronics, Computers, and Communication, administered by the Israeli Academy of Sciences and Humanities.

0890-5401/91 $3.00

An exclusive-read exclusive-write (EREW) PRAM does not allow simultaneous access by more than one processor to the same memory location for read or write purposes. A concurrent-read exclusive-write (CREW) PRAM allows simultaneous access for reads but not writes. A concurrent-read concurrent-write (CRCW) PRAM allows concurrent access for both reads and writes. For the CRCW PRAM, we assume that if several processors attempt to write simultaneously at the same memory location then one of them succeeds but we do not know in advance which one. See (Vishkin, 1983) for a survey of results concerning PRAMs.

Let Seq(n) be the fastest known worst-case running time of a sequential algorithm, where n is the length of the input for the problem at hand. Obviously, the best upper bound on the parallel time achievable using p processors, without improving the sequential result, is of the form $O(\text{Seq}(n)/p)$. A parallel algorithm that achieves this running time is said to have *optimal speed-up* or more simply to be *optimal.* A primary goal in parallel computation is to design optimal algorithms that also run as fast as possible.

Most of the problems we consider can be solved by parallel algorithms that obey the following framework. Given an input of size n the parallel algorithm employs a *reducing* procedure to produce a smaller instance of the same problem (of size $\leqslant n/2$, say). The smaller problem is solved recursively until this brings us below some threshold for the size of the problem. An alternative procedure is then used to complete the parallel algorithm. We refer the reader to (Cole and Vishkin, 1986b) where this algorithmic technique, which is called accelerating cascades, is discussed. Typically, we need to reschedule the processors in order to apply the reducing procedure efficiently to the smaller sized problem. Suppose the input for a problem of size n is given in an array of size n. A natural approach is to compress the smaller problem instance into a smaller array, of size $\leqslant n/2$. This is often done using a standard prefix sum algorithm (it takes $O(\log n)$ time on $n/\log n$ processors to compute the prefix sums for n inputs stored in an array). Thus if we need to reschedule the processors repeatedly it is unclear how to achieve logarithmic time. Sometimes the rescheduling need not be performed very often. For instance, (Cole and Vishkin, 1986a, Cole, 1987) show that for some problems (list ranking and selection) $\log^* n$ reschedulings suffice. Alternatively, one can use a fast random algorithm to perform the rescheduling, or at least an approximate rescheduling. (By approximate rescheduling we mean that we may not be able to partition the work evenly among the processors, but only approximately evenly.) Thus the need for rescheduling does not preclude $O(\log n)$ time optimal random algorithms. Let us mention the main contributions of Part I of this research, the paper (Cole and Vishkin, 1988b). Part I provides an algorithm for performing approximate rescheduling deterministically $O(1)$ time. This is used to solve

a novel scheduling problem. The solution to the scheduling problem leads to a logarithmic time optimal deterministic parallel algorithm for list ranking. In the present paper, a related rescheduling procedure will be one of the tools that leads to a logarithmic time connectivity algorithm which is optimal unless the graph is extremely sparse.

We identify the following *duration-unknown* task scheduling problem. s tasks are given, each of length between 1 and t; the total length of the tasks is bounded by w. (A task can be thought of as a program.) However, we do not know, in advance, the lengths of the individual tasks; in fact, they may vary, depending on the order of execution of the tasks. The problem is to schedule the s tasks on a CRCW PRAM of p processors so that the tasks are completed in $O(t + w/p + \log n/\log^{(2)} n)$ time. This bound applies even if w is not known in advance. This problem is solved in Appendix 1; a similar problem, for EREW PRAMs, was solved in the Part I paper.

We now discuss how to design algorithms that take advantage of this task scheduling algorithm. Given a problem, our job is to design a "protocol" for solving the problem by using a set of short tasks (each of length between 1 and t). This provides an important new opportunity for the designer of a protocol which is based on using the scheduling algorithm: the designer of the protocol need not be concerned to order the execution of the tasks. Such an opportunity for designing parallel tasks, without knowing in advance their lengths, with the guarantee that they will be scheduled efficiently, sounds very promising. However, this opportunity cannot be separated from a considerable difficulty in designing such a protocol: we have no control over the order of execution of the tasks, so we must ensure that the protocol works correctly regardless of the order of execution. We note that this style of protocol design may be useful for distributed systems that are not tightly synchronized; here too, we have to be sure that the protocol works correctly regardless of the order of execution. Part I demonstrates how to design such a protocol for the list ranking problem. In Section 3.3, we demonstrate how to design such a protocol for a problem which occurs in our parallel connectivity algorithm.

The main problem considered in this paper is *graph connectivity*, which is defined as follows:

Input: An undirected graph with n vertices and m edges.

The problem: Find the connected components of the graph.

Results: We obtain the following efficient CRCW PRAM algorithm, *optimal* for $m \geqslant n \log^* n$: On the CRCW PRAM, $T = O(\log n)$ time using $O((n+m)\,\alpha(m, n)/T)$ processors, where $\alpha(m, n)$ is the inverse Ackerman function. The algorithm requires space $O(\min[n^2, mn^{\varepsilon}])$, where ε can be any constant satisfying $0 < \varepsilon < 1$.

In order to shorten this presentation we omitted here a CREW PRAM algorithm: it runs in $T = O(\log^2 n)$ time using $O((n+m)\,\alpha(m, n)/T)$ processors. The algorithm requires $O(n^2)$ space. This algorithm is described in a previous version of this paper (Cole and Vishkin, 1987).

The previous best results for computing connected components are $O(\log n)$ time using $O(m+n)$ processors on the CRCW PRAM (Shiloach and Vishkin, 1982), $O(\log^2 n)$ time using $O((n+m)/\log n)$ processors on the EREW PRAM (Koubek and Krsnakova, 1985), and $O(\log^2 n)$ time using $n^2/\log^2 n$ processors on the CREW PRAM (Chin *et al.*, 1982), or on the CRCW PRAM (Vishkin, 1984). (Kruskel *et al.*, 1989) gave an efficient algorithm for relatively slow times and non-sparse graphs; specifically, they achieved $O((m/p)\cdot(\log n/\log(m/(p^2\log p))) + n\log p/p)$ time and space $O(pn^\varepsilon + m)$, for $\varepsilon > 0$, on the EREW PRAM. (Gazit, 1988) recently gave a randomized connectivity algorithm which runs, with high probability, in logarithmic time using an optimal number of processors.

The literature gives quite a few efficient parallel algorithms for undirected graph problems which essentially reduce a graph problem into the problem of finding a (any) spanning forest in a graph. Fortunately, our connectivity algorithms also find a spanning forest within the same time and processor bounds. This leads, without too much effort, to parallel CRCW algorithms that run in time $O(\log n)$ using $(n\log n + m)/\log n$ processors for the following problems:

1. Finding biconnected components of undirected graphs, using the algorithm of (Tarjan and Vishkin, 1985).

2. Orienting the edges of a connected bridgeless undirected graph so that the resulting directed graph is strongly connected, using the algorithm of (Vishkin, 1985).

3. Ear decomposition and finding *st*-numbering of biconnected graphs, using the algorithm of (Maon *et al.*, 1986).

4. Finding Euler tours in directed and undirected graphs (or determining that they do not exist), using the algorithm of (Atallah and Vishkin, 1984) and a logarithmic time optimal list ranking algorithm. Such algorithms were given in Part I and (Cole and Vishkin, 1989).

With some additional effort, which includes applying the new parallel lowest common ancestor algorithm of (Schieber and Vishkin, 1987) the logarithmic time optimal parallel list ranking of (Cole and Vishkin, 1989), and the parallel expression tree evaluation algorithm of (Cole and Vishkin, 1988a), we extend these logarithmic time optimal speed up results to sparser graphs, where $m = o(n\log n)$. We will not provide the details of these improved algorithms, for it would require us to describe anew parts

of these other papers, and we want to keep this presentation within reasonable length. Also the descriptions of these improvements, particularly for the case $m = o(n \log n)$, are quire tedious, and we doubt whether such lengthly descriptions merit publication.

Previous parallel algorithms for these four problems were given in (Atallah, 1984; Awerbuch *et al.*, 1984; Lovasz, 1985; Savage and Ja'Ja', 1981; Tsin and Chin, 1984).

We make several contributions here. First, we provide a new approach to the connectivity problem. Previous parallel algorithms for the connectivity problem consisted of applying the connectivity computation procedure directly to the whole input graph. Each of these algorithms essentially comprised $O(\log n)$ iterations. Each such iteration "shrank" the input graph further using a number of operations linear in the size of the graph handled by the iteration. While it was possible to decrease the number of vertices from iteration to iteration by a constant factor, it is not known how to achieve a similar reduction in the number of edges. If the original graph was sparse it may become denser following an iteration. Therefore, the resulting connectivity algorithm would need more than a linear number of operations in the worst case. Later, we mention some algorithms that achieved optimality either because the input graph was given by an adjacency matrix or because the family of input graphs was restricted; in each case, a reduction in the number of vertices provides a corresponding reduction in the number of edges. However, these algorithms did not produce optimal connectivity algorithms for general graphs. The new approach finally gives a way to circumvent this problem by applying a connectivity computation procedure to relatively small subgraphs at a time. Thereby, we reduce the total number of operations performed, yielding an algorithm that is optimal except when m is $o(n \log^* n)$. An earlier version of the new connectivity algorithm, which required the same number of operations as here, was given in (Cole and Vishkin, 1986b). However, the time upper bound achieved there was $O(\log n \log^{(2)} n \log^{(3)} n)$.

Second, we exploit two scheduling results to save a time factor of $\log^{(2)} n \log^{(3)} n$ with respect to the algorithm given in (Cole and Vishkin, 1986b).

The first scheduling result is a consequence of a procedure for computing the prefix sums of n numbers, each of $\log n$ bits (the procedure is an optimal CRCW PRAM algorithm that performs $O(n)$ operations in $O(\log n/\log^{(2)} n)$ time). The processor allocation (scheduling) for the connectivity algorithm repeatedly uses this prefix sum procedure. This saves a factor of $\log^{(2)} n$ in the time upper bound, with respect to the standard optimal prefix sum parallel algorithm which runs in $O(\log n)$ time. This new prefix sum procedure is described in (Cole and Vishkin, 1989).

The second scheduling result is a variant of the procedure of the Part I

paper for task scheduling, mentioned above. This saves a factor of $\log^{(3)} n$. The task scheduling procedure is only needed for Step 1, the edge selection.

All these savings together lead to the $O(\log n)$ time optimal connectivity algorithm.

Our third contribution arises because the lack of a logarithmic time optimal parallel connectivity algorithm had been the only obstacle to getting similar results for several other graph problems, as mentioned above. We are not aware of any previous logarithmic time optimal parallel algorithm for a problem on general graphs. As mentioned above, the only known poly-log time optimal parallel algorithms are based on either the assumption that the graph is given by its adjacency matrix or the assumption that the graph is planar. For the first kind of graphs the fastest serial connectivity algorithms use $O(n^2)$ time and the parallel algorithms of (Chin *et al.*, 1982; Vishkin, 1984) achieve $O(\log^2 n)$ time using $n^2/\log^2 n$ processors. These algorithms operate by reducing the size of the adjacency matrix to represent only the shrunken graph at hand. For planar graphs, the main observation needed is that the number of edges can be at most linear in the number of vertices, as follows from Euler's theorem (see Even, 1979)). The parallel algorithms of (Hagerup *et al.*, 1987; Hagerup, 1988) run in $O(\log n \log^* n)$ and logarithmic time, respectively, using an optimal number of processors on planar graphs.

In Section 3 we describe the new CRCW connectivity algorithm. Subsection 3.1 reviews previous work and, thereby, provides motivation for the following subsections. In the beginning of Subsection 3.2, we provide an overview of the other subsections. Appendix 2 gives a parallel union/find algorithm. Preliminary versions of this work appeared as parts of (Cole and Vishkin, 1986b, 1986c).

2. Preliminaries

We give below a useful and simple scheme, due to Brent, for designing parallel algorithms.

Theorem (Brent, 1974). *Consider a PRAM algorithm that comprises k steps, where the ith step uses p_i processors and t_i time. Aside from the time for scheduling processors, such an algorithm can be implemented to run on p processors in time* $\sum_{i=1}^{k} \lceil p_i/p \rceil \cdot t_i \leqslant \sum_{i=1}^{k} (p_i t_i/p + t_i) = (\sum_{i=1}^{k} p_i \cdot t_i)/p + \sum_{i=1}^{k} t_i$.

Proof. On the ith step each actual processor simulates $\lceil p_i/p \rceil$ logical processors in round robin fashion. The result follows easily. ∎

It will be convenient to describe our algorithm using the format suggested by Brent's theorem: namely, the algorithm will comprise a series of steps, and different steps will be described as if different numbers of (logical) processors were available. Of course, as seen in the proof of Brent's Theorem, the logical processors are simulated by actual processors, whose number is fixed throughout the algorithm.

In our algorithms the reallocation of processors will be done in one of two ways: either the reallocation will follow a previously computed pattern, in which case it will take $O(1)$ time, or it will be determined using an optimal parallel prefix sum algorithm, which takes time $O(\log n/\log^{(2)} n)$. In any event, these costs are specifically accounted for in the analysis of the algorithms.

The time bound from Brent's Theorem has the form $(\sum_{i=1}^{k} p_i t_i)/p + \sum_{i=1}^{k} t_i$; this leads us to express the complexity of a step in terms of the pair $(p_i t_i, t_i)$, which we call the (operation, time) complexity. The (operation, time) complexity of a whole algorithm will be $(\sum_1^k p_i t_i, \sum_1^k t_i)$. We will express the complexity of our algorithm in this form.

3. Graph Connectivity

3.1. *Basic Techniques and Previous Work*

We start with a few definitions. Essentially, there exist two parallel polylog time connectivity algorithms. (Hirschberg *et al.*, 1979)(HCS) yields $O(\log^2 |V|)$ time and (Shiloach and Vishkin, 1982) (SV) yields $O(\log |V|)$ time. (We view (Chin *et al.*, 1982; Vishkin, 1984; Wyllie, 1979) as implementations of the HCS algorithm and (Awerbuch and Shiloach, 1983) as an implementation of the SV algorithm.) We review these algorithms briefly and discuss the obstacles to deriving optimal speed-up implementations from them.

Our problem is to compute the connected components of a graph $G = (V, E)$ which is given as follows. *Input form.* Let $V = \{1, ..., n\}$ and $|E| = m$. We assume that the edges are given in a vector of length $2m$. The vector contains first all the edges incident on vertex 1, then all the edges incident on vertex 2, and so on. Each edge appears twice in this vector. We also need the two copies of each edge to be linked; we discuss how to achieve this linking, if it is not provided as part of the input, at the end of Section 3.2.

Definitions (1) A *rooted tree* is a directed graph satisfying:

(a) The undirected graph which is obtained by removing directions from the edges is a tree.

(b) It has a vertex r called the *root* such that there exists a directed path from each vertex to r.

(2) A *rooted star* is a rooted tree in which the path from each vertex to the root comprises (at most) one edge.

The following is common both to the HCS and SV connectivity algorithms and to the new connectivity algorithm presented here. At each step during the algorithms each vertex v has a pointer $D(v)$ through which it points to another vertex or to no vertex. One can regard the directed edge $(v, D(v))$ as a directed edge in an auxiliary graph, called the *pointer graph.* Initially, for each vertex v, $D(v)$ points to no vertex and therefore the initial pointer graph consists of only the vertices, but has no edges. The pointer graph keeps changing during the course of the algorithms. However, at each step of each of these three algorithms the pointer graph consists of rooted trees. It will be convenient to refer to a set of vertices comprising a tree as a *supervertex.* Sometimes, we identify a supervertex with the root of its tree. No confusion will arise. As the algorithm proceed, the number of trees (supervertices) decreases. This is achieved by (possibly simultaneous) *hooking* operations. In each hooking a root r of a tree is 'hooked' onto a vertex v of another tree (that is, $D(r) := v$). A careful look at each of these connectivity algorithms reveals that:

1. Each such hooking is performed only after the algorithm "identified" an edge connecting a vertex in the supervertex of r with a vertex in the supervertex of v. Let us call such a connecting edge the *causing edge* of its hooking. We illustrate this notion of causing edge in the description of the HCS algorithm given below.

2. (*The spanning forest property.*) For each supervertex, consider the collection of the causing edges that connect pairs of its vertices. This collection forms a spanning tree of the vertices comprising the supervertex. Thus, the collection of the causing edges, throughout each of these connectivity algorithms, forms a spanning forest of the input graph.

The trees are also subject to a *shortcut* operation. That is, for every vertex v of the tree,

if $D(D(v))$ is some vertex (as opposed to no vertex)
then $D(v) := D(D(v))$.

The shortcut operation (approximately) halves the height of a tree. Shortcuts do not introduce cycles into the pointer graph, as can be readily verified. Simultaneous hookings are performed in each of these algorithms in such a way that no cycles are introduced into the pointer graph.

The algorithms also use the following graph. Each edge (u, v) in the input graph induces an edge connecting the supervertex containing u with the supervertex containing v. The graph whose vertices are the supervertices and whose edges are these induced edges is called the *supervertex graph.*

At the end of each of these algorithms the vertices of each connected component form a rooted star (which is, in particular, a single supervertex) in the pointer graph. As a result, a single processor can answer a query of the form "do vertices v and w belong to the same connected component?" in constant time.

The HCS parallel connectivity algorithm works in $O(\log n)$ iterations. Upon starting an iteration each supervertex is represented by a rooted star in the pointer graph. Each root hooks itself onto a minimal root which is adjacent to it in the supervertex graph. In case two roots are hooked on one another, we cancel the hooking of the smaller (numbered) root. As a result several rooted stars form a rooted tree; the root of one of these rooted stars becomes the root of the new tree. An iteration finishes with $O(\log n)$ shortcuts. It remains to identify the causing edges in this algorithm. In the HCS algorithm a root hooks itself onto a minimal adjacent root. There might be more than one edge connecting the two supervertices, but the algorithm selects precisely one of these edges. The selected edge, which then induces the hooking in the course of the computation, is the causing edge of the hooking.

The SV parallel algorithm also works in $O(\log n)$ iterations. Unlike the HCS algorithm: (i) An iteration of SV takes constant time, and (ii) The pointer graph at the beginning of an iteration is a collection of rooted trees (which are not necessarily stars). In principle, an iteration comprises the following steps.

(1) Each rooted star is hooked onto a smaller vertex that is adjacent to some vertex of its supervertex (if there is any such smaller vertex).

(2) Consider the rooted stars that did not hook and were not hooked upon in step (1); each such rooted star is hooked onto a vertex that is adjacent to some vertex of its supervertex.

(3) Shortcuts.

The algorithm employs a processor for each vertex and each edge of the graph. This amounts to $n+m$ processors. (Shiloach and Vishkin, 1982) shows that the total height of "still active" trees decreases by a factor of at least 1/3 per iteration, implying that only $O(\log n)$ iterations are needed and, therefore, the algorithm runs in $O(\log n)$ time. The algorithm does not achieve optimal speed up.

3.2. *The New Algorithm*

Our new algorithm for finding connected components runs in time $O(\log n)$. It achieves optimal speed-up for graphs with $m \geqslant n \log^* n$, and almost optimal speed up in general. The algorithm has two parts.

The second (and more basic) part is an algorithm for relatively dense graphs ($m \geqslant n \log n \log^{(3)} n$); we call this the *main* connectivity algorithm. (Implicitly, we are assuming $n \geqslant 16$, to ensure $\log^{(3)} n \geqslant 1$.) The main connectivity algorithm can be viewed as a two level improvement in efficiency relative to the SV algorithm. The first level achieves optimal speed-up with a parallel time of $O(\log n \log^{(3)} n)$; the second level achieves optimal speed-up with a time of $O(\log n)$. The second level is asymptotically better; however, it involves the use of expander graphs and consequently the "big oh" notation hides considerably larger constants. The first level is described later in this section and the second level is described in Section 3.3.

The first part is a reduction procedure which (essentially) reduces the general case to the dense case. The reduction procedure requires $O((m+n)\,\alpha(m, n))$ operations and $O(\log n)$ time. It is described in Section 3.4. We remark that for $m \geqslant n \log^* n$, for example, it is optimal. Incidentally, the reduction procedure does not involve expander graphs or similar constructs. The presence of the $\alpha(m, n)$ term is due to the use of a new parallel union/find algorithm, which is of interest in its own right.

Our algorithm uses a number of parameters that strictly speaking are not integers, but which nonetheless we treat as integers (for instance, $\log n$). To justify this we view all parameters as being rounded up to the nearest integer power of 2. In more complex expressions, namely products and ratios (such as $\log n/\log^{(2)} n$), we round each of the basic terms separately ($\log n$ and $\log^{(2)} n$ here) and then compute the expression with these rounded terms; in our expressions the result is always a non-negative integer power of 2. The rounding to the next larger power of two, rather than to the next larger integer, is particularly convenient in Section 3.4. Also, it is convenient to interpret log 1 to be 1.

High-Level Description of the Main Connectivity Algorithm

We state the opportunity and main difficulty in obtaining an optimal speed-up algorithm from the SV algorithm. In the SV algorithm, a processor is standing by each edge. The opportunity is that for each processor there is at most one step during the whole algorithm during which its edge is used for hooking; the difficulty is that we do not know in advance when this step comes.

The main connectivity algorithm applies the SV connectivity algorithm. The key to obtaining an optimal algorithm is the following strategy: The SV algorithm is applied only to selected subsets of the edges of the graph,

the subsets changing as the algorithm proceeds. More specifically, the $O(\log n)$ iterations of the SV algorithm are now divided into $O(\log \log n)$ phases. At the start of each phase new edges subsets are selected. Roughly speaking, each phase includes about as many iterations of the SV algorithm as all the preceding phases put together. Below, we characterize each phase by an input/output relation. The parameter used for this characterization is denoted by the somewhat awkward notation *d_squared*. The remark below justifies this awkwardness.

Input: For each supervertex, its constituent vertices have $\geq d\_squared$ incident edges from G, where *d_squared* is an integer, $d\_squared \geq 2$, and an edge with both endpoints in the supervertex is counted twice.

Output: For each supervertex, its constituent vertices have $\geq d\_squared^{1.5}$ incident edges from G.

Remark: The remark on rounding applies to these input and output parameters *d_squared* and $d\_squared^{1.5}$ in the following way. Inductively, *d_squared* was realized as a power of 2. A key parameter is $d = (d\_squared)^{1/2}$. It is realized simply as the next largest power of 2 of the real number $(d\_squared)^{1/2}$. $d\_squared^{1.5}$ is realized as $d \cdot d\_squared$, which is a power of two. $d\_squared^{1.5}$ becomes *d_squared* of the next phase. Finally, note the fact $d\_squared \leq d^2 \leq 2d\_squared$. This fact will be used later in the text.

Before outlining the implementation of a phase, we need a few definitions for classifying the edges of G with respect to the supervertex graph. Given a supervertex graph, an edge of G is *redundant* (with respect to the supervertex graph) if both its endpoints lie in the same supervertex. An edge is an *outedge* if it is not redundant. If several outedges connect the same pair of supervertices, one of these outedges is chosen to be the *actual* outedge; the other outedges are called *duplicate* outedges. (The rule for choosing the actual outedge is specified below, in the detailed description of a phase.) The *degree* of a supervertex v is defined to be the number of actual outedges incident on v. Also, in each phase, we classify the supervertices according to whether they already satisfy the output condition; thus we define a supervertex to be *large* if it is known to have at least $d \cdot d\_squared$ incident edges, and to be *growing* otherwise. A phase comprises the following three steps.

Step 1. Select an edge set such that

(i) For each growing supervertex of degree $\leq d$, all the actual outedges are selected.

(ii) For each growing supervertex of degree $\geq d$, exactly d actual outedges are selected.

Step 2. Run the SV connectivity algorithm for $\lfloor \log_{3/2} d \rfloor + 1$ iterations on the graph induced by the edges selected in Step 1. (This is the number of iterations needed to guarantee that the supervertices being created all satisfy the output condition for the phase, as we show later in the detailed analysis of Step 2.)

Step 3. This step is applied only to those new supervertices (rooted trees from Step 2) that will be growing in the next phase. This includes each new supervertex with fewer than $(d \cdot d\_squared)^{1.5}$ incident edges. For each such supervertex, we form an adjacency list of its incident edges (this is needed for Step 1 of the next phase). We call this *contracting* the supervertex (for one can view this step as contracting the tree for a new growing supervertex to a single node).

Later in Section 3.2 we give a detailed description and analysis of each step.

For the purposes of the analysis, it is useful to guarantee that each vertex has degree an integer multiple of $\log n \log^{(3)} n$. To achieve this, we add up to $\log n \log^{(3)} n$ self loops per vertex; each instance of a self loop (v, v) is recorded exactly once on v's adjacency list (rather than twice, once for each endpoint). We add at most m edges (recall that, by assumption, $m \geqslant n \log n \log^{(3)} n$). In fact, the self loops do not have to be added; it suffices to pretend that they are present for the purposes of the analysis. Also, each self loop is defined to contribute one incident edge to the supervertex to which it belongs.

Observations. a. There are at most $3m$ incident edges in the graph (where each instance of each edge is counted, i.e., an edge is counted once in each adjacency list in which it occurs).

b. It is safe to set $d\_squared = \log n \log^{(3)} n$ initially, since each vertex has degree $\geqslant \log n \log^{(3)} n$. After $O(\log \log n)$ phases there will be only one supervertex comprising all the vertices in the graph. (Actually, we could choose a smaller initial value for $d\_squared$. The present implementation of Step 3.1.1.2 requires $d\_squared \geqslant \log n / \log^{(2)} n$ to achieve an optimal algorithm; with an alternative implementation, we could reduce the initial value of $d\_squared$ somewhat. However, as we are trying to make $d\_squared$ grow, we might as well initialize it with the largest possible value.)

c. In Step 1, $O(m/d)$ edges are selected. (Since the input to the phase has $\leqslant 3m/d\_squared \leqslant 6m/d^2$ supervertices).

d. Consider the components of the graph comprising the supervertices and the edge set selected in Step 1, above. Each growing supervertex of degree less than d has all its incident actual outedges selected. Thus, each

component either includes a supervertex with at least d actual outedges, or it includes a large supervertex, or there is just one component. As we show (in the detailed description of Step 2) each supervertex formed in Step 2 either comprises $\geq d$ input supervertices (and so has $\geq d \cdot d\_squared$ incident edges), or contains a large input supervertex (and so has $\geq d \cdot d\_squared$ incident edges), or comprises all the vertices.

e. Contracting all supervertices with fewer than $(d \cdot d\_squared)^{1.5}$ incident edges guarantees that any supervertex that is not contracted in step 3 is *large* for the next phase. If a supervertex has sufficiently many incident edges it may be large for several consecutive phases.

The procedure for reducing the general connectivity problem to the case $m \geq n \log n \log^{(3)} n$, given in Section 3.4, has the same structure as the main connectivity algorithm. However, Steps 1 and 3 are implemented somewhat differently.

High Level Analysis

Step 1. Its analysis is deferred to the detailed description of this step.

Step 2. Per phase, it processes $O(m/d)$ edges for $O(\log d)$ steps. Thus, per phase, it performs $O((m/d) \log d)$ operations in $O(\log d)$ time. Over the whole algorithm this is $O(m \log^{(2)} n/(\log n \log^{(3)} n)^{1/2}) = O(m)$ operations and $O(\log n)$ time.

Step 3. For those new supervertices that will be large in the next phase, not only is it unnecessary to contract them, it is too expensive (in time and operations) to do so. In the current phase we allocate $\Theta(\log d)$ time and $\Theta(m \log d/\log n)$ operations for performing contractions. We call this resource allocation the *increasing budget* for time and operations. The increasing budget may not be big enough to complete the contraction of a larg supervertex in the current phase. However, as later phases have larger increasing budgets, a later phase, in which the supervertex is no longer large, will be able to do the contraction.

There are other parts of Step 3 whose cost is not covered by the increasing budget. These parts will be dealt with when the detailed description of Step 3 is given.

We have already introduced one budget used in the analysis. Altogether, the analysis uses four kinds of budgets (for time and number of operations):

(1) An *increasing* budget (for each phase). The increasing budget for each of the two items (namely, time and number of operations) increases from phase to phase; specifically, it is $O(\log d)$ time and $O(m \log d/\log n)$ operations.

(2) A *fixed* budget (for each phase). The fixed budget is the same for each phase: $O(\log n \log^{(3)} n/\log^{(2)} n)$ time and $O(m/\log^{(2)} n)$ operations.

(3) A *general-pool* budget (for the whole algorithm): $O(\log n \log^{(3)} n)$ time and $O(m)$ operations.

(4) *Miscellaneous* budgets (for each phase). These are specified as they are needed (one miscellaneous budget was used in the analysis of Step 2 above).

The *total* budget for the whole algorithm is the sum of these budgets over all the phases.

The fixed and general pool budgets are used in the analysis of Step 1, the edge selection. We note that the general-pool budget is independent of the budget for any individual phase; in other words, the edge selection procedure has an interesting amortized complexity analysis.

Actually we do not need to use the budgets in our analysis below. The reason for specifying the budgets, and referring to them in the analysis, is to emphasize the different (time, operation) combinations used in the various substeps of the algorithm.

Next, we describe in detail and analyze each step in turn.

The Edge Selection (Step 1)

Here, we give a procedure for edge selection that performs $O(m)$ operations in time $O(\log n \log^{(3)} n)$ over the course of the whole algorithm. In Section 3.3 we show how to reduce this to $O(\log n)$ time, without changing this operation count.

In this step the processors are reallocated every $\Theta(\log n/\log^{(2)} n)$ time units, using a parallel prefix sum algorithm. We do not know how to reallocate the processors faster; consequently, we will process the edges in units of size $\log n/\log^{(2)} n$, called *blocks*. Thus, at the start of the algorithm, the edges incident on each vertex are divided into blocks of $\log n/\log^{(2)} n$ edges each (for recall that by the addition of self loops we ensured that each vertex has degree a multiple of $\log n \log^{(3)} n$). In general, the edges incident on each supervertex are divided into blocks of $\log n/\log^{(2)} n$ edges, but with possibly some incomplete blocks (i.e., blocks having too few edges). However, there will always be at most $3m/\log n$ incomplete blocks (this is certainly true initially, and this invariant will be maintained in Step 3). The division into blocks may change from phase to phase. More precisely, new blocks are created by combining and repartitioning blocks that belong to the same growing supervertex. The total number of blocks at hand is always bounded by $3m \log^{(2)} n/\log n$, as we will show in the description of Step 3.

Data Structures. $T(v)$ is the name of the supervertex currently containing vertex v of G (T is the *vertex table*).

$ACTUAL(u, v)$ is an $n \times n$ array; it records if an edge connecting supervertex u and supervertex v has been found in the current phase. Specifically, $ACTUAL(u, v)$ comprises a pointer to an edge plus a timestamp; the timestamp is a phase number. (We explain at the end of this section how to implement $ACTUAL$ in space $O(\min[n^2, mn^\varepsilon])$ for any fixed $\varepsilon > 0$; evaluating $ACTUAL(u, v)$ takes time $O(1/\varepsilon)$).

The block headers (i.e., names) are stored in a single array, with the blocks for each growing supervertex occupying a contiguous portion of the array. The edges in each block are stored in a linked list.

The growing supervertex headers are stored in an array. Each supervertex header records the span occupied by its block headers. This array also contains headers for each growing supervertex from a previous phase that is part of a large supervertex; more precisely, each such growing supervertex remains in this array until it becomes part of a new larger growing supervertex. We call such growing supervertices *out-of-date* growing supervertices.

We say growing supervertex v is *active* for the current iteration of Step 1.1 (below) of the edge selection procedure unless either we have found d actual outedges incident on v, or we have checked all the edges incident on v. A block is active if its supervertex is active. Let b_A (resp. b'_A) be the number of active blocks at the start (resp. end) of the current iteration of Step 1.1.

In the procedure below, the active blocks (or rather, their headers) are kept in an array with blocks belonging to the same supervertex being contiguous. This procedure assumes that $p = 3m/\log n \log^{(3)} n$ processors are available. (This choice for p arises because we are aiming for an optimal algorithm, and because even if we have $\Theta(m/\log n)$ processors available, the procedure below will still require $\Theta(\log n \log^{(3)} n)$ time, as we justify later. The choice of the constant 3 is unimportant; it simplifies some constants in the analysis.)

Step 1.1. **while** $b_A > 0$ **do**
begin
1.1.1. Process every $\lceil b_A/p \rceil$th block in the array of active blocks.
Processing a block using a single processor: For each edge $e = (i, j)$ in the block, determine whether it is redundant, duplicate, or actual. In the first two cases eliminate the edge from further consideration; in the last case add the edge to the set of selected edges. Proceed as follows.
(a) Let $u = T(i)$, $v = T(j)$. If $u = v$, the edge is redundant. If not, continue with step (b).

(b) Check $ACTUAL(u, v)$. If an edge is recorded with the current timestamp then e is a duplicate outedge. Otherwise continue with step (c).
(c) Edge e attempts to write its address and the current phase number to $ACTUAL(u, v)$. If the write is successful e is selected (e is an actual outedge); otherwise e is a duplicate outedge.

1.1.2. For each growing supervertex, determine if it is still active. (This step is easily performed using a parallel summation algorithm with respect to the block headers.) Then compress the (headers of the) still active blocks to the start of the active block array. (Detecting the active blocks is facilitated by marking the first and last block for each growing supervertex; it then merely requires a parallel prefix sum computation to separate the active blocks.)

end

Step 1.2. For each growing supervertex, among the edges selected in Step 1.1, we will select d (or all, if $<d$ are available). The selected edges will be placed in an array of size $O(m/d)$. First, a parallel prefix sum computation with respect to the block headers determines how many edges each block provides, and for each block provides a range of serial numbers for the edges contributed by that block. Then the blocks that might contribute edges are processed in the same order as in Step 1.1 to place the selected edges in the array of size $O(m/d)$.

Analysis. We note that each iteration of Step 1.1 takes time $O(\log n/\log^{(2)} n)$ and performs $O(m/(\log^{(2)} n \log^{(3)} n))$ operations. We show that over the whole algorithm, these iterations perform $O(m)$ operations in time $O(\log n \log^{(3)} n)$.

We first present a special case of the analysis; it provides the intuition and an overview for the general case. So, for now, suppose that all the blocks are complete and all the supervertices have an integer multiple of $\lceil b_A/p \rceil$ blocks. Then we can partition the iterations into two cases.

Case 1. $b'_A \leqslant (1/2) b_A$—a *reducing* iteration. (recall b_A and b'_A are the number of blocks active, respectively, at the start and at the end of the iteration). Below, we show that, per phase, there are $O(\log^{(3)} n)$ reducing iterations (Claim 1). Thus, per phase, the reducing iterations use $O(\log n \log^{(3)} n/\log^{(2)} n)$ time, and perform $O(m/\log^{(2)} n)$ operations (to obtain this multiply $O(\log^{(3)} n)$ iterations by $O(m/(\log^{(2)} n \log^{(3)} n))$ operations). This is charged to the fixed budget. Over the whole algorithm this is $O(\log n \log^{(3)} n)$ time and $O(m)$ operations.

Case 2. $b'_A > (1/2) b_A$. Then at least half the blocks processed belong to still active supervertices; we call these the *productive* blocks. Each edge in a productive block is either eliminated or found to be an actual outedge.

We partition non reducing iterations into eliminating and selecting iterations.

Case 2a. At least half the edges processed, among the productive blocks, were eliminated—an *eliminating* iteration. The number of edges eliminated in an eliminating iteration is at least $3m/(4 \log^{(2)} n \log^{(3)} n)$ (since for at least half the blocks, namely the productive blocks, at least half the edges were eliminated). But at most $3m$ edges can be eliminated (each edge in each adjacency list in which it is present). Thus there are at most $4 \log^{(2)} n \log^{(3)} n$ eliminating iterations over the course of the algorithm. The cost of these iterations ($O(\log n \log^{(3)} n)$ time and $O(m)$ operations) is charged to the general pool budget.

Case 2b. More than half the edges processed, among the productive blocks, were actual outedges (i.e., not case 2a)—a *selecting* iteration. The operations of a selecting iteration are charged to the actual edges found which belong to supervertices that remain active; we call these edges *charged edges.* Clearly, there are at least $3m/(4 \log^{(2)} n \log^{(3)} n)$ charged edges per selecting iteration (the actual edges provided by the productive blocks). We show that there are fewer than $6m/d$ charged edges over the course of the phase (Claim 2). Thus there are at most $\lfloor 8 \log^{(2)} n \log^{(3)} n/d \rfloor = O(1)$ selecting iterations per phase (since $d \geqslant (\log n \log^{(3)} n)^{1/2}$; in fact, for large enough n there are 0 selecting iterations per phase). Per phase, these iterations cost $O(\log n/\log^{(2)} n)$ time and $O(m/\log^{(2)} n \log^{(3)} n))$ operations. This is charged to the fixed budget. Over the whole algorithm this is $O(\log n)$ time and $O(m/\log^{(3)} n)$ or $O(m)$ operations.

But rounding makes everything messier. The problems are caused by two sorts of blocks, which we call *obstructive* blocks. First, incomplete blocks, of which there are at most $3m/\log n$, as asserted at the start of the description of Step 1. Second, blocks which were processed in excess disproportion to the supervertex size, for supervertices that become inactive; that is, if v has $k\lceil b_A/p \rceil + l$ blocks, where $l < \lceil b_A/p \rceil$, and $k+1$ of v's blocks are processed, and v becomes inactive, then with one of v's processed blocks we are unable to associate a further $\lceil b_A/p \rceil - 1$ of v's blocks that became inactive; call this block the *excess* block (we arbitrarily choose the excess block to be a specific block among v's blocks, say the block of smallest index in the active block array, processed in this iteration). There is at most one excess block per supervertex over the course of a phase; that is, at most $3m/d\_$ *squared* excess blocks, per phase. In order to incorporate obstructive blocks into the analysis we change Case 2 and introduce Case 3.

New Case 2. At least a quarter of the blocks processed are productive.

New Case 2a. At least half the edges processed, among the productive blocks, were eliminated—an *eliminating* iteration. Clearly, we at most double the number of eliminating iterations as compared to the previous analysis (since each eliminating iteration need only eliminate half as many edges as before). So the cost of the eliminating iterations is still ($O(\log n \log^{(3)} n)$ time and $O(m)$ operations).

New Case 2b. More than half the edges processed, among the productive blocks, were actual outedges (i.e., not New Case 2a)—a *selecting* iteration. Again, we at most double the number of selecting iterations per phase. So, over the whole algorithm, the cost of the selecting iterations is still $O(\log n)$ time and $O(m/\log^{(3)} n) = O(m)$ operations.

Case 3. Otherwise; that is, $b'_A > 1/2\, b_A$ and fewer than a quarter of the blocks processed are productive—a *removing* iteration. We show that at least a quarter of the processed blocks are obstructive (Claim 3). As argued above, in the current phase there can be at most $3m/\log n + 3m/d\_squared \leqslant 6m/\log n$ obstructive blocks. Since $3m/(\log n \log^{(3)} n)$ blocks are processed in each iteration, and as the iteration is removing only if at least a quarter of the processed blocks are obstructive, we conclude that there are at most $8 \log^{(3)} n$ removing iterations per phase. Thus, the removing iterations perform $O(m/\log^{(2)} n)$ operations in time $O(\log n \log^{(3)} n/\log^{(2)} n)$ per phase. This is charged to the fixed budget. Over the whole algorithm, this is $O(m)$ operations and $O(\log n \log^{(3)} n)$ time.

So, over the whole algorithm, all four types of iterations perform $O(m)$ operations in $O(\log n \log^{(3)} n)$ time.

REDUCING ITERATIONS. CLAIM 1. *There are only* $O(\log^{(3)} n)$ *reducing iterations in a phase.*

Proof of Claim 1. After $\log^{(3)} n + \log^{(4)} n$ reducing iterations the number of active blocks decreases by a factor of at least $\log^{(2)} n \log^{(3)} n$. Thus, in the next iteration a processor will be assigned to each (presently) active block and following this iteration there will be no active blocks left. ▮

SELECTING ITERATIONS. CLAIM 2. *There are fewer than* $6m/d$ *charged edges over the course of the phase.*

Proof of Claim 2. Consider a charged edge e and the iteration t in which it was charged. Let e be in a block of supervertex v. At the end of iteration t the supervertex v remained active. Thus fewer than d actual edges incident on v had been discovered at the end of iteration t. We conclude that the last charged edge, in this phase, for supervertex v is at most

the $d-1$st charged edge for supervertex v. As there are at most $6m/d^2$ supervertices in this phase the claim follows. ▮

REMOVING ITERATIONS. CLAIM 3. *In a removing iteration at least a quarter of the processed blocks are obstructive.*

Proof of Claim 3. Recall that in a removing iteration $b'_A > 1/2\, b_A$ and fewer than a quarter of the blocks processed are productive. Consider a processed block B; let B belong to supervertex v. There are four possibilities.

Case A. B is productive. Since the iteration is removing, fewer than a quarter of the blocks fit this case.

Case B. B becomes inactive, but B is not the excess block for v. Since the iteration is not reducing, fewer than half the blocks fit this case (for recall we can associate a further $\lceil b_A/p \rceil - 1$ of v's blocks with B, and all these associated blocks become inactive).

Case C. B is the excess block for v, if any.

Case D. B is incomplete.

Cases C and D include only obstructive blocks. Clearly, in a removing iteration, Cases C and D must include at least a quarter of the processed blocks. ▮

It is clear that the complexity of Step 1.2 is dominated by that of Step 1.1.

Comment. With more processors, but still using $O(m/\log n)$ processors, it is possible to perform all the eliminating, selecting, and removing iterations in $O(\log n)$ time; however, this does not apply to the reducing iterations. In each phase, there may be $\Theta(\log^{(3)} n)$ reducing iterations, even if $m/\log n$ processors are used. As mentioned above, this guides the choice of $p = \Theta(m/(\log n \log^{(3)} n))$. Thus, varying the parameters will not speed up Step 1 by more than a constant factor, at least so long as we aim for an optimal algorithm.

Also, this is why the main algorithm requires $m \geqslant n \log n \log^{(3)} n$. In the first phase, in each iteration of Step 1.1, at least $\Theta(n \log n/\log^{(2)} n)$ operations are performed (to process one block per vertex); so in the first phase, the reducing iterations may perform $\Theta(n \log n \log^{(3)} n/\log^{(2)} n)$ operations. With a uniform implementation of the edge selection in each phase this implies that $\Theta(n \log n \log^{(3)} n)$ operations may be performed. For an optimal algorithm, we therefore require $m \geqslant n \log n \log^{(3)} n$.

Implementation of the Array ACTUAL. We show how to implement array *ACTUAL* in $O(mn^\varepsilon)$ space, for any fixed $\varepsilon > 0$. Since an alternative implementation in $O(n^2)$ space is obvious, we conclude that array *ACTUAL* can be implemented in $O(\min(n^2, mn^\varepsilon))$ space. Our description below ignores the timestamps attached to each edge.

We use two arrays, A and B. Array A is of size $n \times n^\varepsilon$ and array B is of size $m \times n^\varepsilon$. Suppose the ith edge of the input graph connects supervertices u and v, where $1 \leqslant i \leqslant m$. We show how to store this edge in these arrays using a single processor. This parallel procedure will take at most $1/\varepsilon$ steps. Each of u and v is a number between 1 and n. Suppose, without loss of generality, that (the number) u is smaller than (the number) v. Let $v_1 v_2 \cdots v_{1/\varepsilon}$ be the representation of v with respect to base n^ε. (If $1/\varepsilon$ is not an integer take instead $\lceil 1/\varepsilon \rceil$.)

Step 1. Write i in location $A(u, v_1)$. Observe that concurrent writes may occur. Suppose that edge j_1 was actually written into $A(u, v_1)$. If $i = j_1$ then we have finished storing edge i. Otherwise,

Step l, $1 < l \leqslant 1/\varepsilon$. Write i in location $B(j_{l-1}, v_l)$. Again, concurrent writes may occur. (The following is not required in Step $1/\varepsilon$). Suppose that edge j_l was actually written into $B(j_{l-1}, v_l)$. If $i = j_l$ then we have finished storing edge i. Otherwise, proceed to Step $l + 1$.

It is easy to verify that all the edges being inserted will be stored by the end of Step $1/\varepsilon$. Also, given two vertices u and v, it takes at most $1/\varepsilon$ steps to find out whether there exists an edge connecting u and v, using the information in arrays A and B.

The array *ACTUAL* is also used to link the two copies of each edge at the start of the algorithm. We proceed in two stages. First for each edge (u, v) with $u < v$ we insert the edge into the array *ACTUAL*. Next, for each edge (u, v) with $u > v$ we search for the edge (v, u), which is already in the array *ACTUAL*. The two copies of the edge are then readily linked. It is clear this requires $O(1/\varepsilon)$ time and $O(1/\varepsilon \cdot m)$ operations.

Step 2. As stated above, in the dth phase we apply the SV connectivity algorithm for $O(\log d)$ time to the graph specified by the supervertices and the edges selected in Step 1. This algorithm was described in Section 3.1. Actually, we need to make one minor change to the algorithm: only growing supervertices participate in an active way in the algorithm; the large supervertices, or rather their constituent out-of-date growing supervertices, simply provide nodes for hooking onto. This implies that when the pointer of a growing supervertex v points to an out-of-date growing supervertex, then v no longer performs pointer jumping; i.e., henceforth v participates only in a passive way in the SV algorithm. Otherwise the growing supervertices perform the algorithm in the standard way. Note that the array of

growing and out-of-date growing supervertices provides the vertex set for the SV algorithm here.

The SV connectivity algorithm requires that the edges be accessible via an array whose size is twice the number of the edges and that each edge appears there once in each of its two directions. It is easy to achieve this. In Step 1.2, an edge is placed in the selected edge array in both of its directions. (Even if an edge was selected by both its endpoints, no damage will be cause by the redundant edges).

The SV algorithm maintains the following invariant: Let T be a tree constructed by the algorithm in t steps, and let l be the length (in edges) of the longest path from a leaf to the root in tree T; then either $|T| \geqslant (3/2)^{t-1} \cdot l$, for $t \geqslant 1$, or T is a spanning tree of a maximal connected component. Thus after $\lfloor \log_{3/2} d \rfloor + 1$ steps, any tree built by the SV algorithm either will contain at least d vertices, or will be a maximal connected component.

We deduce that after $\lfloor \log_{3/2} d \rfloor + 1 = O(\log d)$ iterations, any tree built by the algorithm, comprising only growing supervertices, includes either at least d old supervertices or all the old supervertices, as claimed in Observation d (immediately following the high level description of the main connectivity algorithm above); the only other possibility is that the tree contains a large supervertex.

The processor allocation for Step 2 is straightforward: simply assign one processor to each edge in the array computed in Step 1.2. Step 2 performs $O(m \log d/d)$ operations in $O(\log d)$ time (using the formulation of [SV-82], Step 2 uses $md/d\_squared = O(m/d)$ processors and $O(\log d)$ time). This is charged to a miscellaneous budget. Over the whole algorithm this is $O(m)$ operations and $O(\log n)$ time.

Step 3. For each new supervertex (a tree constructed in Step 2) that will be growing in the next phase, Step 3 forms an adjacency list of its edges, divided into blocks. Basically, first, we form an Euler circuit of each tree constructed in Step 2; second, for each tree, we combine the adjacency lists of the nodes in the tree with the help of the Euler circuits; third, for each block of an adjacency list L such that L is not too long, we recognize that the block belongs to L. This recognition is performed by applying Wyllie's (1979) list ranking algorithm to the adjacency lists. In using Wyllie's algorithm, we perform pointer juming for each unit in the lists being processed; over the course of the algorithm, there will be $\Theta(\log n)$ pointer jumping steps per unit processed. In order to obtain an optimal algorithm we need to process the adjacency lists in units of size $\Omega(\log n)$ edges.

This motivates us to create *clusters.* For each supervertex, its edges are partitioned into clusters of at most $\log n$ edges each; each cluster comprises up to $\log^{(2)} n$ blocks. Initially, for each vertex, each cluster contains exactly

$\log^{(2)} n$ blocks (recall that each vertex initially has a degree that is an integer multiple of $\log n \log^{(3)} n$). Each cluster can only lose edges (from the edge elimination); a cluster never gains edges. However, the edges within a cluster may be repartitioned among its blocks; thus a block may lose and gain edges, but only from within its cluster. There will be at most one incomplete block per cluster. There are at most $3m/\log n$ clusters (since there are at most $3m$ edges initially); hence there are at most $3m/\log n$ incomplete blocks and at most $3m \log^{(2)} n/\log n$ blocks. We keep an array of cluster headers, with clusters belonging to the same growing supervertex being contiguous. Each cluster header records the span of the headers of its blocks in the block header array (it will always be the case that a cluster's blocks are contiguous in the block header array). Also, for each supervertex, we keep its clusters in a circular linked list.

The input for Step 3 comprises:

(a) For each processor, the ordered list of the blocks it processed in Step 1.

(b) The array of clusters. This array also includes dummy clusters, whose role will become clear later. There are $O(m/\log n)$ dummy clusters (Claim 4).

(d) The array of growing supervertices.

Next we outline the procedure for Step 3.

Step 3.1. For each supervertex (tree) created in Step 2, form a single linked list of its clusters. This will require the introduction of dummy clusters, and will be based on an Euler Tour of each tree.

Step 3.2. Separate the new growing and new large supervertices. That is, for each cluster recognize the type of its new supervertex, and if it is growing, in addition, find out the name of the new growing supervertex. This step is performed by applying Wyllie's list ranking algorithm to the lists formed in Step 3.1.

Step 3.3.

Step 3.3.1. Removes those edges eliminated by the edge selection in Step 1 and form new blocks so that there is at most one incomplete block in each cluster.

Step 3.3.2. Rearrange the arrays of block headers and cluster headers so that for each growing supervertex, its associated blocks (resp. clusters) are contiguous. Also, update the array of growing supervertices.

Step 3.4. Update the vertex table.

Below we alternately use several patterns for the assignments of processors to jobs. One pattern is called *the pattern of Step* 1: each processor processes one block at a time, in the same order as in Step 1 (this is easily obtained from item (a) of the input to Step 3). The other patterns will be described as they are used.

Step 3.1.

Step 3.1.1. In order to carry out Step 3.1. we need to compute two numberings for each edge that became a causing edge in this phase (recall the causing edges are the edges that induce the hookings in Step 2). First, we number all the causing edges with a single serial numbering. Second, for each cluster, we number its causing edges with a serial numbering. Observe that while a growing supervertex v seeks to hook itself onto only one other supervertex in Step 2, several of its incident edges may become causing edges that hook the supervertex at their other endpoint onto v.

In order to simplify the exposition we neglected to add to Step 2 an instruction for marking each copy of a causing edge (immediately after it is used for hooking). So we assume that each causing edge is so marked. Whenever, in Step 3, we refer to a causing edge, we mean a copy of a causing edge introduced in Step 2 of the present phase.

The numberings are computed as follows. In Step 3.1.1.1 we assign a processor to each causing edge. In Step 3.1.1.2 we assign serial numbers to the causing edges in each block. In Step 3.1.1.3 we assign the two desired serial numbers to each causing edge.

Step 3.1.1.1. By means of a prefix summ algorithm with respect to the array of edges used in Step 2 we assign one processor to each of the two copies of each causing edge; this takes $O(\log n/\log^{(2)} n)$ time and $O(m/d)$ operations, per phase. There are at most $12m/d^2$ copies of causing edges, since there are at most $6m/d^2$ growing supervertices each providing at most one causing edge and each causing edge has two copies.

Step 3.1.1.2. For each block, we compute both the number of its causing edges and the serial number of each of them. For each copy of a causing edge, its processor traverses the list of edges in the block of this copy, till it comes to the front of the list (which has a pointer to the block header); since there may be several processors traversing a block only the processor of the first causing edge in the block (using the original order of the block) remains to take care of the block. It first assigns serial numbers to each causing edge in the block and later writes their total number to a variable associated with the block header. This takes $O(\log n/\log^{(2)} n)$ time and $O(m \log n/(d^2 \log^{(2)} n))$ operations per phase.

Step 3.1.1.3. For each cluster, we visit its block headers in turn (i.e., serially) and assign to each block a range for the serial numbers of its

causing edges relative to the other causing edges of the cluster. This takes $O(\log^{(2)} n)$ time and $O(m \log^{(2)} n/\log n)$ operations per phase.

Then, by means of a prefix sum computation with respect to the block headers, we assign to each block a range for the serial numbers of its causing edges relative to the whole set of causing edges. This takes $O(\log n/\log^{(2)} n)$ time and $O(m \log^{(2)} n/\log n)$ operations per phase.

Now, using the processor assignment pattern computed in Step 3.1.1.1 (recall that it provides a separate processor to each causing edge) we assign the two serial numbers to each causing edge. This takes $O(1)$ time and $O(m/d^2)$ operations per phase.

Step 3.1.2. Our goal here is to provide, for each new supervertex S, a (circular) linked list that goes through all the clusters in S. Recall that at the beginning of a phase the clusters of each (old) supervertex were arranged in a circular linked list. Let C be a cluster. Suppose C has h causing edges. In this circular list of clusters, we replace cluster C by a list comprising cluster C plus h dummy clusters; each of the dummy clusters represents one of the h causing edges introduced in the present phase. Each old supervertex now has a circular list comprising all the clusters it had previously plus some additional dummy clusters. (Note that to order the dummy clusters associated with cluster C we need the second of the serial numbers computed in Step 3.1.1.3).

CLAIM 4. *The number of dummy clusters created during the whole algorithm is bounded by* $2(n-1)$ *which is* $O(m/\log n)$.

Proof of Claim 4. Recall that the input vertices and causing edges form a spanning forest; thus there are at most $n-1$ causing edges. ∎

Next, we show how to form a single circular list for each new supervertex. This new list will include all the clusters from the old supervertices that form the new supervertex. To achieve this we apply an idea of (Atallah and Vishkin, 1984) for "stitching" the circular lists at the causing edges. We note that a causing edge has a copy (which is a dummy cluster) in two of these circular lists. Each copy has a successor in its own list. In parallel, *we make the successor of each copy of each causing edge the successor of the other copy of the same causing edge.* An argument as in (Atallah and Vishkin, 1984) shows that this indeed gives a single circular list for each new supervertex.

The number of operations required is proportional to the number of causing edges, which is $O(m/d^2)$. The time is $O(1)$.

It is convenient to place the dummy clusters introduced in this phase into the array of clusters. To do this we need to assign a serial number to

each such dummy cluster; we simply add the first serial number associated with the corresponding causing edge to the current size of the cluster array. Per phase this takes $O(1)$ time and $O(m/d^2)$ operations.

Henceforth, when we refer to dummy clusters, we intend all the dummy clusters that are present, and not just those created in the current phase. (In Step 3.2 we will see why dummy clusters from previous phases might be present).

Step 3.2. Our goal is to separate new growing from new large supervertices. It will be helpful to attach a distinct identifier to each cluster. Thus, each dummy cluster, on creation, is given as identifier the pair $(|V|+u, v)$, where (u, v) is the causing edge that caused the dummy cluster to be created. Likewise, each actual (non-dummy) cluster is given the identifier (u, v), where (u, v) is the first edge on the edge list of its first block.

It is helpful to note that if a supervertex has γ actual clusters it can have at most 2γ dummy clusters. For the causing edges in the supervertex form a spanning tree of its constituent vertices and so there is one fewer causing edge than constituent vertices. It remains to note that each vertex contributed at least one actual cluster, while each causing edge produced just two dummy clusters.

A supervertex will be large in the next phase if it has at least $(d \cdot d\_squared)^{1.5}$ incident edges. Let $\gamma_d = (d \cdot d\_squared)^{1.5}/\log n$. Thus, for each new growing supervertex, the circular list o clusters, actual and dummy, comprises fewer than $3\gamma_d$ nodes (at most γ_d actual clusters and fewer than $2\gamma_d$ dummy clusters). It is convenient to redefine a large (resp. growing) new supervertex to be one with at least (resp. fewer than) $4\gamma_d$ nodes in its circular list of clusters. This implies that a new large supervertex has at least $4/3(d \cdot d\_squared)^{1.5}$ incident edges (since at least 1/3 of the nodes are actual clusters and each actual cluster contains $\log n$ edges), and a new growing supervertex has fewer than $4(d \cdot d\_squared)^{1.5}$ incident edges. For each of our lists we determine whether it represents a large or growing new supervertex by the following computation.

(a) We iterate at each element, in parallel, the following basic (doubling) operation $\beta_d = \log(2\gamma_d)$ times.

1. The new identifier of the element is the (lexicographic) minimum between its own identifier and the identifier of its successor.

2. Doubling (i.e., $D(v) := D(D(v))$, where initially $D(v) =$ successor(v)).

Following this computation each element holds the minimum identifier among its own (original) identifier and the (original) identifier of its $2^{\beta_d} - 1$ original successors.

(b) We check at each element whether its minimum identifier and the minimum identifier of its (present) successor are equal.

We observe that the cyclicity of each list implies that if the minimum identifiers of some element and its present successor are equal then: (i) this minimum identifier must be the minimum identifier of their list. (ii) the list contains at most $2^{\beta_d+1}-1$ elements and therefore the supervertex is growing.

(c) We apply β_d+1 parallel doublings in order to "broadcast" this minimum. In each list in which no minimum identifier was found, nothing is being broadcast and each node in such a list can conclude that it is part of a new large supervertex.

(d) Using β_d+1 parallel doublings, for the lists of growing supervertices, we shortcut over, and thereby discard, the dummy clusters.

(e) Next we give serial numbers, starting from one, to the actual clusters of each growing supervertex, as follows. Using β_d+1 parallel doublings, for the list of growing supervertices, we rank each actual cluster with respect to the element whose identifier is minimum in its list.

The processor allocation for Step 3.2 is to provide one processor to each cluster and dummy cluster ($O(m/\log n)$ processors). Thus, per phase, Step 3.2 takes $O(\log d)$ time and $O(m \log d/\log n)$ operations.

Step 3.3.

Step 3.3.1. The goal is to remove those edges eliminated by the edge selection in Step 1 and to form new blocks so that there is at most one incomplete block per cluster.

Step 3.3.1.1. For each block processed in Step 1 we remove those edges eliminated in Step 1, forming a list of the remaining edges. Also, for each block, we record the number of edges still present. The processor allocation for this step is given by the pattern of Step 1; its complexity is dominated by that of Step 1.

Step 3.3.1.2. For each cluster we form a list of its blocks. We partition each such list into two sublists. The first sublist comprises the non-empty blocks that were processed in Step 1 and incomplete blocks (there is at most one incomplete block for each cluster). The second sublist contains the complete blocks that were not processed in Step 1. Empty blocks are discarded. This separation is performed in parallel for each cluster by a sequential scan of the block headers in the cluster. Next, for each cluster, for each first list, we compute the prefix sums of the block sizes. This step uses $O(\log^{(2)} n)$ time and $O(m \log^{(2)} n/\log n)$ operations, per phase.

Step 3.3.1.3. For each sublist of the first type, we form new blocks, as

follows. Using the prefix sums, we determine the new block boundaries. (This requires scanning the edges in the blocks containing such boundaries.) Then we append each list of edges to the end of the list for the preceding block. For each cluster, the edges are now divided into complete blocks plus at most one incomplete block. This takes $O(\log n/\log^{(2)} n)$ time and, per block processed, $O(\log n/\log^{(2)} n)$ operations.

The processor allocation for this step follows the pattern of Step 1, except that in addition we need to process each old incomplete block, of which there is at most one per cluster; but this just requires an additional allocation of one processor per cluster, which is straightforward. Thus the complexity of this step is dominated by that of Step 1.

Step 3.3.1.4. For each new block, it remains to place its header in the portion of the block array belonging to its cluster. For each new block B, we choose an old block B' being removed from the same cluster; B's header will take the place occupied by B''s header. B' can be chosen according to the following rule: it is the first old block which overlaps with B. Next, for each cluster, we compress its block headers to a contiguous portion of the block array, by means of a prefix sum computation with respect to the array of block headers. We also record, for each cluster, the span occupied by its present block headers. This takes $O(\log n/\log^{(2)} n)$ time and $O(m \log^{(2)} n/\log n)$ operations per phase.

Step 3.3.2. Each new supervertex is presently represented by a circular list of clusters; the list for large supervertices include dummy clusters, while the lists for growing supervertices do not (Step 3.2). During this step, the array of cluster headers is reordered so that clusters belonging to the same new growing supervertex are in contiguous locations. The array of blocks headers is reordered correspondingly.

We compute the new location for each cluster header as follows. From part (e) of Step 3.2, we have already determined, for each new growing supervertex, the number of clusters it contains. This number is placed in the 'first' cluster for the supervertex; every other cluster in a new growing supervertex is assigned the number zero. The clusters in new large supervertices are all assigned the number one. By performing a prefix sum computation over these numbers with respect to the array of clusters, we obtain the new location of each first cluster of a new growing supervertex and of each cluster of a new large supervertex. Finally, each cluster in a new growing supervertex computes its location relative to the first cluster in its new growing supervertex. This step uses $O(\log n/\log^{(2)} n)$ time and $O(m/\log n)$ operations per phase.

Next, we rearrange the block headers so as to match the new cluster ordering. To do this, each cluster determines how many blocks it has; then

by means of a prefix sum computation with respect to the cluster headers it determines the new locations of its block headers. Then, each block header is given its new location by its cluster. Finally, the block headers are relocated simultaneously. Per phase, this takes $O(\log n/\log^{(2)} n)$ time and $O(m \log^{(2)} n/\log n)$ operations.

The array of growing supervertices is readily updated with the help of a parallel prefix sum computation with respect to the array of clusters; it sums the number of "first" clusters, including the "first" clusters for out-of-date growing supervertices. Per phase, this takes $O(\log n/\log^{(2)} n)$ time and $O(m/\log n)$ operations.

Step 3.4. Initially, we associate with each vertex v its 'first' cluster C_v (chosen to be the minimum among its clusters when ordered by their identifiers). C_v will be responsible for updating $T(v)$. A growing supervertex is given the vertex name associated with its 'first' cluster.

To update the vertex table T, if C_v is part of a growing supervertex S, it performs the following operation. Let w be the vertex name associated with S (in Step 3.2(c) each cluster belonging to S learnt this name); C_v performs the assignment $T(v) := w$.

The processor assignment for this step is to provide one processor to each cluster; then this step takes $O(1)$ time and $O(m/\log n)$ operations per phase.

Analysis. We have shown that, per phase, the complexity of Step 3 is dominated by the sum of three components:

(i) the complexity of Step 1,

and complexities of

(ii) $O(\log n/\log^{(2)} n)$ time and $O(m/\log^{(2)} n)$ operations,

(iii) $O(\log d)$ time and $O(m \log d/\log n)$ operations.

(To verify this, it suffices to recall that $d \geqslant (\log n \log^{(3)} n)^{1/2}$. Summing over all the phases, we obtain that over the whole algorithm, the complexity of Step 3 is bounded by the complexity of Step 1, plus a complexity of $O(m)$ operations and $O(\log n)$ time. Hence Step 3 performs $O(m)$ operations in $O(\log n \log^{(3)} n)$ time.

We conclude that

THEOREM 3.2: *For $m \geqslant \log n \log^{(3)} n$ there is an optimal connectivity algorithm in the CRCW PRAM model that performs $O(m+n)$ operations in time $O(\log n \log^{(3)} n)$ on $(m+n)/\log n \log^{(3)} n$ processors.*

3.3. *Connectivity in Logarithmic Time and Optimal Speed Up*

Here we show how to implement the connectivity algorithm so that it runs in time $O(\log n)$. As in Subsection 3.2, we assume that $m \geqslant n \log n \log^{(3)} n$. But here we assume that the number of processors available is $p = m/\log n$. The basic structure of the algorithm remains unchanged. The algorithm consists of $O(\log^{(2)} n)$ phases each with the same input/output definition as before. Steps 2 and 3 will remain unchanged.

Here we reduce the block size to $\log n/(\log^{(2)} n)^2$, for we will redistribute the processors following every $\Theta(\log n/\log^{(2)} n)^2)$ time units (the redistribution is done using an approximate scheduling scheme, not by a prefix sum algorithm). Again, there may be at most one incomplete block per cluster. We introduce a new structure here: groups; for each cluster, we partition its blocks into groups of $\log^{(2)} n$ blocks, with at most one incomplete group per cluster. In each phase, the groups are created at the start of Step 1, by means of a prefix sum algorithm with respect to the block headers. The increase in the number of blocks requires us to reanalyze Step 3. However, a careful examination of the analysis shows that it still uses $O(m)$ operations and $O(\log n)$ time over the whole algorithm (recall that the only reason the previous version of Step 3 used $\Theta(\log n \log^{(3)} n)$ time was the processing of blocks in the pattern of Step 1; but here such processing will only take $O(\log n)$ time).

The only step we need to redesign is Step 1, the edge selection. To carry out the edge selection we create the following subgoal: for each group, process its blocks one by one until either $\log n/(\log^{(2)} n)^2$ actual outedges have been found or all its blocks have been processed. Having achieved this subgoal, there are fewer than $d(\log^{(2)} n)^2/\log n$ not fully processed groups for each still active supervertex (since for each such supervertex fewer than d actual outedges have been found, and for each not fully processed group $\log n/(\log^{(2)} n)^2$ actual outedges have been found). Thus there remain at most $6m(\log^{(2)} n)^3/d \log n$ active blocks (the number of active blocks remaining is derived by multiplying $6m/d^2$ supervertices by $d(\log^{(2)} n)^2/\log n$ not fully processed groups by $\log^{(2)} n$ blocks per group); these are readily processed in a further $O(\log n/(\log^{(2)} n)^2)$ time and $O(m \log^{(2)} n/d)$ operations, which is $O(m \log^{(2)} n/(\log n \log^{(3)} n)^{1/2})$ operations, since $d \geqslant (\log n \log^{(3)} n)^{1/2}$. Note also that we need to use a prefix sum algorithm to allocate the processors to the active blocks. This prefix sum computation will perform a number of operations which is linear in its input size. The input size is equal to the number of blocks, which is $O(m(\log^{(2)} n)^2/\log n)$. The time for the prefix sums is $O(\log n/\log^{(2)} n)$. These costs are charged to the fixed budget. To find the charge for the whole algorithm, we multiply each of these time counts and operation counts by $O(\log^{(2)} n)$ phases and get $O(\log n)$ time and $O(m)$ operations.

It remains to describe how to achieve the subgoal. This is achieved using the approximate task scheduling of the Part I paper, or rather the following slight modification of this scheme: We are given s tasks, each of duration between 1 and t, and of total duration w; using p processors the tasks can be performed in $O(t + \log n/\log^{(2)} n + w/p)$ time. This bound applies even if w is not known in advance. (Note that the sequencing of the tasks is not predetermined; in fact, according to the sequencing of tasks, each individual task may have a varying length, but always in the range $[1, t]$; also, the sum of the lengths of the tasks is always bounded by w.) The procedure for executing such a collection of tasks is given in Appendix 1; here we show how to cast the edge selection problem in terms of such tasks.

For each group its processing comprises a task. The length of a task is equal to $\log n/(\log^{(2)} n)^2$ times the number of blocks processed by this task; thus the length is between $\log n/(\log^{(2)} n)^2$ and $\log n/(\log^{(2)} n)$. For any given phase, the total length of the tasks is unknown (for it depends on how many edges are eliminated); for phase i denote the total length of its tasks by w_i. The number of processors at hand is $m/\log n$.

LEMMA 3.3.1. *The total work performed by the tasks over all the phases is* $O(m)$.

Proof. In each phase, the work performed per group is at most the number of edges eliminated, plus at most $2 \log n/(\log^{(2)} n)^2$ (this term accounts for the actual edges found; although only $\log n/(\log^{(2)} n)^2$ actual edges per group are sought, since each group is processed in units of a block, for each group, we may find an additional $\log n/(\log^{(2)} n)^2 - 1$ actual outedges). So, per phase, the total work performed is proportional to the number of edges eliminated plus $m/\log^{(2)} n$ (since there are at most $3m \log^{(2)} n/\log n$ groups at hand). Since each edge can be eliminated at most once in each direction and since there are $O(\log^{(2)} n)$ phases, the total work performed over all the phases is $O(m)$. ∎

We conclude

THEOREM 3.3. *There is a CRCW PRAM algorithm for computing the connected components of a graph that performs* $O(m+n)$ *operations in* $O(\log n)$ *time using* $(m+n)/\log n$ *processors, if* $m \geqslant n \log n \log^{(3)} n$.

Comment. In fact, we only need $m \geqslant n \log n/\log^{(2)} n$ for the result of Theorem 3.3 to hold, as can be seen by checking the analysis of Step 3 and the new Step 1.

3.4. *The Reduction Procedure*

We show how to reduce the connectivity problem for the case $m < n \log n \log^{(3)} n$ to the case $m \geqslant n \log n \log^{(3)} n$. The reduction procedure will build supervertices each of which has at least *d_squared* incident edges (from the input graph) with $d\_squared \geqslant 4 \log n \log^{(3)} n$ (the reason for the constant 4 is that there are at most $4m$ edges present, counting each occurrence of an edge, as we will see shortly). Following the application of the reduction procedure we rename all the edges with their current supervertex endpoints, obtaining a *reduced* graph. We can then apply the connectivity algorithm of Section 3.3 to the reduced graph; the supervertices of the reduced graph are the vertices for the connectivity algorithm. (This ensures that for the reduced graph the number of vertices is bounded by $m/(\log n \log^{(3)} n)$).

It is convenient to assume that each vertex has degree at least 1. This is since any remaining vertex constitutes a separate component. All such vertices can be readily removed in a preprocessing phase using $O(m+n)$ operations and $O(\log n)$ time. Following such a preprocessing phase we can assume that $n \leqslant 2m$. The reduction procedure uses the same basic procedure as the connectivity algorithm, that is, we have a series of phases parameterized by d or *d_squared*. Initially $d\_squared = \max\{2, m/n\}$; this is justified by introducing up to $2m$ self loops. Thus, counting each occurrence of an edge in an adjacency list, there are at most $4m$ edges present.

Now we proceed through only $O(\log^{(3)} n)$ phases. In each phase, $d\_squared < 4 \log n \log^{(3)} n$; the last phase increases *d_squared* to a value of at least $4 \log n \log^{(3)} n$. Each phase is divided into 3 steps, as before. The role of each step is identical. In fact, Step 2 is implemented just as before. Steps 1 and 3 need to be modified. We describe the changes to each step in turn.

The data structures and algorithms become somewhat simpler here for the blocks and clusters can have the same size. Thus we will identify blocks and clusters in this section, and refer to them interchangeably. While not the only choice, as it appears the simplest, we choose the block size to be d edges. (Roughly, any value of block size between $\log^2 d$ and $d^2/\log^2 d$ can be used; in fact both these bounds can be extended a little. Of course, other parameters then need to be changed accordingly. We discuss the constraints on the block size in the comment below.) The data structures are mostly the same as in the main algorithm (except for the obvious simplifications and changes to their sizes). However, we need to make several changes to the way in which the supervertices are presented. First, we need to label each supervertex that has been identified as comprising a component of the input graph; such a supervertex is called a *complete* supervertex. The

remaining supervertices are divided into growing and large supervertices as before. Second, we will divide the large supervertices into pseudo-growing supervertices; the pseudo-growing supervertices are treated in the same way as growing supervertices, but there is no lower bound on their size. There are at most $5m/d\_squared$ growing and pseudo-growing supervertices, and at most $4m/d\_squared$ growing supervertices (Claim 5). Each growing or pseudo-growing supervertex will contain at most one incomplete block. Thus, there are at most $4m/d + 5m/d\_squared \leqslant 9m/d \leqslant 12m/d$ blocks (for $d \leqslant d\_squared$). The pseudo-growing supervertices replace the out-of-date growing supervertices which are not present in the reduction algorithm.

Comment. In the reduction procedure we cannot use one size of blocks for all the phases, as in the main algorithm. There are three constraints. First, in Step 1, for each block that is processed, we perform a number of operations proportional to the upper bound on the block size (multiplied by a factor of α here, as we will see). For each incomplete block, the "wasted" effort is proportional to the difference between the upper bound on the block size and the actual block size. Summing over all incomplete blocks in the first phase (of which there may be as many as n), we obtain that the block size should be upperbounded by $O(m/n)$ to ensure that the wasted effort is $O(m)$ (multiplied by a factor of at most α). If $m = O(n)$, this implies that initially we need a block size of $O(1)$. Second, in Step 3, we may perform $\Theta(\log d)$ operations per block during the pointer jumping. This implies a lower bound of $\Theta(\log d)$ on the block size; but this only ensures a bound of $O(m)$ operations per phase. A lower bound of roughly $\log^2 d$ on the block size suffices to provide a linear operation bound for Step 3 over the whole algorithm. Third, in Step 1, as already noted, processing a block takes time proportional to the upper bound on the block size multiplied by α; obviously, this should not exceed $\Theta(\log n)$, and maintaining an upper bound of $\Theta(\log n/\log^{(2)} n)$ on the running time for processing a block simplifies the analysis; this implies an upper bound of $\Theta(\log n/(\log^{(2)} n\, \alpha))$ on the block size; in turn, this implies an upper bound of roughly $d^2/\log^2 d$ on the block size, since d^2 may grow as large as roughly $\log n \log^{(3)} n$.

Step 1. Basically Step 1 proceeds as in Section 3.2. One important change is than an operation $x := T(v)$ will take time $O(\alpha(m, n))$ rather than $O(1)$. Henceforth, we will use α to denote $\alpha(m, n)$. (See Step 3.4 below and Appendix 2 for a description of how to maintain the vertex table.)

We divide the computation of Step 1.1 into two parts for the purposes of the analysis. The first part is Step 1.1.1. This part is most easily understood if we assume that $p = 12m/(d \log d)$ processors are available (we simply simulate this by the actual number of processors which is only $12m\alpha/\log n$);

so $12m/d \log d$ blocks are processed in each iteration. (We justify this choice of p in the comment below.) Step 1.1.1 requires $O(d\alpha)$ time and performs $O(m\alpha/\log d)$ operations. The second, and less interesting part, is Step 1.1.2 (determining if a supervertex is still active). Per iteration, this requires time $O(\log n/\log^{(2)} n)$ and $O(m/d)$ operations. So an iteration (of steps 1.1.1 and 1.1.2) uses $O(m\alpha/\log d)$ operations and $O(\log n/\log^{(2)} n)$ time.

As before, we divide the iterations into reducing, eliminating, selecting, and removing iterations.

There are only $O(\log^{(2)} d)$ reducing iterations per phase (the reasoning being as before: after $\log^{(2)} d$ reducing iterations there is a processor standing by each block of each active supervertex). The reducing iterations, per phase, require $O(m\alpha \log^{(2)} d/\log d)$ operations and $O(\log n \log^{(2)} d/\log^{(2)} n)$ time. Thus, over the whole procedure ($O(\log^{(3)} n)$ phases), the reducing iterations perform $O(m\alpha)$ operations in $O(\log n \log^{(3)} n/\log^{(2)} n)$ or $O(\log n)$ time.

An eliminating iteration eliminates $\Theta(m/\log d)$ edges using $\Theta(m\alpha/\log d)$ operations and $O(\log n/\log^{(2)} n)$ time. As $\log d = O(\log^{(2)} n)$ there are $O(\log^{(2)} n)$ eliminating iterations. Thus, over the whole algorithm, the eliminating iterations perform $O(m\alpha)$ operations in $O(\log n)$ time.

A selecting iteration will charge its operations to the following *charged edges*: those actual outedges in productive blocks. In an iteration there are $\Theta(m/\log d)$ charged edges. Also, at most $O(m/d)$ edges can be charged during the phase. Thus there are $O(\log d/d)$ which is $O(1)$ selecting iterations per phase, and in fact, 0 selecting iterations per phase as soon as d exceeds an appropriate constant. Thus there are only $O(1)$ selecting iterations over the whole algorithm. Each iteration uses $\Theta(m\alpha/\log d)$ operations and $O(\log n/\log^{(2)} n)$ time. They perform a total of $O(m\alpha)$ operations in $O(\log n/\log^{(2)} n)$ or $O(\log n)$ time.

A removing iteration processes at least $3m/(d \log d)$ obstructive blocks; but there are at most $4m/d\_squared$ incomplete blocks and $4m/d\_squared$ excess blocks which are processed in the current phase (at most one incomplete block and one excess block for each growing supervertex), that is, at most $8m/d\_squared$ obstructive blocks; hence there are at most $O((\log d)/d)$ removing iterations per phase, which is $O(1)$ removing iterations over the course of the algorithm. Each iteration uses $\Theta(m\alpha/\log d)$ operations and $O(\log n/\log^{(2)} n)$ time. They perform a total of $O(m\alpha)$ operations in $O(\log n/\log^{(2)} n)$ or $O(\log n)$ time.

Comment. We use $\Theta(m/(d \log d))$ logical processors here for the following reason. (Actually, we may simulate these logical processors by fewer actual processors later). Suppose $p = m/(df(n, d))$, where f is some function of n and d. There are several constraints on f.

(1) We cannot afford more than $\Theta(\log^{(2)} n)$ eliminating iterations if we are to remain within an $O(\log n)$ time bound; if all eliminating iterations occur in one phase each of these eliminating iterations must eliminate at least $\Theta(m/\log^{(2)} n)$ edges. But, using $m/(df(n, d))$ processors, we eliminate $\Theta(m/f(n, d))$ edges in each eliminating iteration; this forces $f(n, d) = O(\log^{(2)} n)$.

(2) Each iteration takes $\Theta(m/d)$ operations for Step 1.1.2. There may be up to $\Theta(\log f(n, d))$ reducing iterations per phase. So, per phase, Step 1.1.2 alone may require $\Theta((m/d)\log f(n, d))$ operations for the reducing iterations. So we need $\sum (\log f(n, d))/d = O(1)$ where the sum is over those values of d that occur in the reducing algorithm.

(3) Now let us consider Step 1.1.1 in the reducing iterations; per phase, this step may perform $\Theta(m\alpha \log f(n, d)/f(n, d))$ operations. So we need $\sum (\log f(n, d))/f(n, d) = O(1)$, where the sum is over the same values of d as before.

The simplest solution to these constraints appears to be $f(n, d) = \log d$.

Step 1.2. For each growing supervertex, among the edges selected in Step 1.1. select d (or all, if $< d$ are available). The selected edges are placed in an array of size $O(m/d)$. Again, the complexity of Step 1.2 is dominated by the complexity of Step 1.1.

Step 3. Many of the substeps are similar to those for the main algorithm. Steps 3.1–3.4 all have the same goals as before. Step 3.3, besides its previous tasks, also prepares new larger blocks for the next phase here.

Step 3.1.

Step 3.1.1. It is identical, but its analysis changes. Step 3.1.1.2 now uses $O(d)$ time and $O(m/d)$ operations per phase, and Step 3.1.1.3 now uses $O(\log n/\log^{(2)} n)$ time and $O(m/d)$ operations. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

Step 3.1.2. This step is essentially identical. The complexity per phase is as before. Over the whole algorithm it is $O(\log n)$ time and $O(m)$ operations.

Step 3.2. As before, the goal is to separate the growing and the large supervertices. We use a different method here. In addition, because of the need to create larger blocks for the next phase, each large supervertex will be divided into *pseudo-growing supervertices.* We will still retain the structure of the large supervertices. This will be made precise later.

Suppose we are given a set of circular lists stored in an array of size h. Before describing the rest of the algorithm, it is useful to recall that there

is a maximal independent set algorithm for rings (circular lists) that uses $O(\log h/\log^{(2)} h)$ time and $O(h)$ operations (Cole and Vishkin, 1989).

We iterate the following computation $3\log d + 2$ times.

(i) Apply the maximal independent set algorithm to the lists constructed in Step 3.1 and then shortcut over the nodes not placed in the independent set.

(ii) Compress the selected nodes to the start of the array.

(The initial array is a copy of the array of block headers.) Any list that is reduced to a single node is defined to represent a new growing supervertex. Note that a node of a maximal independent set is followed, in its circular list, by one or two nodes which are not in the independent set and then by a node of the maximal independent set. Thus a list that is reduced to a single vertex contains at least $(4/3)\,d^3$ edges (since at most 2/3 of the nodes are dummy blocks, and each actual block contains at least one edge). For the remaining new supervertices, the new large supervertices, we define each sublist that has been reduced to a single node to be a new *pseudo-growing supervertex*. There is no lower bound on the size of a pseudo-growing vertex since it may have been built entirely of dummy blocks.

Per phase, the complexity of this step is $O(\log n \log d/\log^{(2)} n)$ time and $O(m \log d/d)$ operations. Over the whole algorithm this is $O(\log n)$ time and $O(m)$ operations.

Step 3.3. We perform the two substeps of the main algorithm out of order. First we perform the previous Step 3.3.2. Then we perform a modified Step 3.3.1. Finally, we introduce a new Step 3.3.3 to create new larger blocks.

Step 3.3.2. This step is identical. We treat all the blocks as parts of growing supervertices (this includes the pseudo-growing supervertices). Per phase, it takes time $O(\log n/\log^{(2)} n)$ and $O(m/d)$ operations. Over the whole algorithm this step uses $O(\log n)$ time and $O(m)$ operations.

Step 3.3.1. This step is essentially identical. However, the goal is modified to produce at most one incomplete block for each growing or pseudo-growing supervertex. Here, the role of the clusters in the main algorithm is taken by the growing and pseudo-growing supervertices. This entails one change to the procedure. The previous Step 3.3.1.2 (separating the blocks into two lists) is now performed by a pointer jumping procedure that follows in the footsteps of the current Step 3.2. As before, the complexity of Step 3.3.1 has two components. One component is dominated by the complexity of Step 1, while the other becomes $O(\log n \log d/\log^{(2)} n)$ time and $O((m \log d)/d)$ operations per phase. Over the whole algorithm, the second component totals $O(\log n)$ time and $O(m)$ operations.

Step 3.3.3. Here, we create new larger blocks for the next phase. For each growing or pseudo-growing supervertex, we combine sets of old blocks (each of d edges, except possibly for one incomplete block) to form new blocks (each of $((d \cdot d\_squared)^{1.5})^{1/2}$ edges, except possibly for one incomplete block). This step uses a prefix sum computation with respect to the old array of blocks in order to identify the new blocks. A second prefix sum computation is used to compress the new block headers to the start of the block header array; in addition, for each new growing and pseudo-growing supervertex, we record the new span of its blocks. Per phase, the step has complexity $O(\log n/\log^{(2)} n)$ time and $O(m/d)$ operations. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations.

Step 3.4. In this step, we update the vertex table. The new growing supervertices (including pseudo-growing supervertices) will be placed in an array, and each old growing supervertex is given the name of its new supervertex.

Handling the new pseudo-growing supervertices presents some difficulties, for the blocks of an old supervertex may be split over several new pseudo-growing supervertices. Our solution is to identify each old supervertex (growing or pseudo-growing) with the first edge in its 'first' block; each old supervertex is defined to belong to the new supervertex (growing or pseudo-growing) that contains this first edge. This definition causes no difficulty: any vertex in a large supervertex S is identified with a pseudo-growing supervertex that is a part of S. Thus, in Step 3.3, for each old supervertex, we need to keep track of the new block to which its first edge belongs; but this is readily done. It remains to place the new complete supervertices in one array and the remaining supervertices (growing or pseudo-growing) in a second array. This is achieved with the use of a prefix sum computation with respect to the array of block headers.

We then update the vertex table using the method of Appendix 2. Appendix 2 provides a limited UNION–FIND procedure. This procedure performs alternate phases of UNIONs and FINDs. There are two classes of sets, changing and fixed sets. A fixed set has no further UNIONs applied to it, while a changing set may have further UNIONs applied to it; a changing set can subsequently become fixed. Further, following i UNION phases the bound on the number of changing sets is $cn/2^i$, where n is the initial number of sets and c is some constant. If there are n sets originally and $O(m)$ FINDs are performed, then the UNION–FIND procedure performs its UNIONs using $O(n\alpha)$ operations and $O(\log n)$ time, while each FIND requires $O(\alpha)$ operations and time, but as many FINDs as desired can be performed in parallel. This UNION–FIND procedure requires that the new sets be presented in 2 arrays, one for the changing sets and one for the fixed sets, and that the old changing sets be presented in a third array,

with each old changing set having a pointer to the new set of which it forms a part.

For our application we make the following assignments. The sets correspond to the supervertices. The fixed sets are exactly the complete supervertices. The changing sets correspond to the remaining growing and pseudo-growing supervertices that "contain" at least one input vertex in the following sense. A new pseudo-growing or growing supervertex contains those old growing and pseudo-growing supervertices that are identified with it; the closure of this relation over the phases identifies those vertices contained in each current pseudo-growing supervertex (the reason for this definition is that a pseudo-growing supervertex may contain no vertices; however, a growing supervertex will always contain at least one vertex). It is straightforward to identify and mark those pseudo-growing supervertices that contain no vertex when the new pseudo-growing supervertices are formed. Rather than place the non-complete supervertices in one array, as stated above, we separate them into two arrays, one for the growing or pseudo-growing supervertices containing at least one vertex, and one for the remaining pseudo-growing supervertices. This computation is performed using a prefix sum algorithm. The UNION–FIND procedure is provided with two of these arrays: the array of complete supervertices, and the array of growing and pseudo-growing supervertices containing at least one vertex. We choose $c = 2.5$. So we need that at the end of the ith phase the number of growing and pseudo-growing supervertices is bounded by $2.5n/2^i$. But this upper bound is immediate from Claim 5 for the values of d and $d\_squared$ at least double from phase to phase and are initially at least 2 and $\max\{2, m/n\}$, respectively.

Per phase, exclusive of the work done by the procedure of Appendix 2, the complexity of this step is $O(\log n/\log^{(2)} n)$ time and $O(m/d)$ operations. Over the whole algorithm, this is $O(\log n)$ time and $O(m)$ operations. The procedure of Appendix 2 performs $O(n\alpha + m)$ operations in $O(\log n)$ time.

Analysis. As before, the complexity of Step 3 is bounded by the complexity of Step 1, plus a complexity of $O(\log n)$ time and $O(m)$ operations, plus the complexity of the Appendix 2 procedure ($O(n\alpha + m)$ operations and $O(\log n)$ time).

CLAIM 5. *The number of old growing and pseudo-growing supervertices is at most* $5m/d\_squared$, *of which at most* $4m/d\_squared$ *are growing supervertices. Also, there are at most* $m/d\_squared$ *new pseudo-growing supervertices with fewer than* $d\_squared$ *edges. Likewise, the number of new growing and pseudo-growing supervertices is at most* $5m(d \cdot d\_squared)$, *of which at most* $4m/(d \cdot d\_squared)$ *are growing supervertices. Also, there are at most* $m/(d \cdot d\_squared)$ *new pseudo-growing supervertices with fewer than* $d \cdot d\_squared$ *edges.*

Proof of Claim 5. We prove the claim by induction on the (implicit) phase number. Clearly the first part of the claim is true initially. Also, if the second part of the claim is true then the first part is true at the start of the next phase (since the new value of *d_squared* is just $d \cdot d\_squared$). To prove the inductive step it suffices to show that the second part of the claim follows from the first part. We start by showing the bound on the number of new pseudo-growing supervertices with fewer than $d \cdot d\_squared$ edges. So consider a new pseudo-growing supervertex S which has fewer than $d \cdot d\_squared$ edges. When S was constructed in Step 3.2, it contained at least $4d^3$ blocks, of which fewer than d^3 contained any edges, that is, it contained at least $3d^3$ dummy blocks. But at most one dummy block can be provided by each old pseudo-growing supervertex and then only if it has no edges. (For in Step 3.3.1 of the previous phase, if any, we ensured that each pseudo-growing supervertex had at most one incomplete block, which was a dummy block only if the supervertex had no edges. If this is the first phase then there are no pseudo-growing supervertices.) By the inductive hypothesis there are at most $m/d\_squared$ old pseudo-growing supervertices with fewer than $d\_squared$ edges; the remaining dummy blocks are dummy blocks created in this phase, of which there are at most $8m/d\_squared$ (at most 2 for each old growing supervertex); this is a total of at most $9m/d\_squared$ dummy blocks. So there are at most $9m/(3d^3 d\_squared) \leqslant m/(d \cdot d\_squared)$ (recall $d \geqslant 2$) new pseudo-growing supervertices with fewer than $d \cdot d\_squared$ edges. The remaining new pseudo-growing supervertices and all the growing supervertices each contain at least $d \cdot d\_squared$ edges; since there are at most $4m$ edges at hand, the rest of the claim now follows easily. ∎

Constructing the Output Graph. The reduction procedure yields a graph with at most $m/(\log n \log^{(3)} n)$ supervertices, large and growing. It remains to rename each edge with its current supervertex endpoints. Then we can apply the main algorithm. We proceed as follows.

1. For each block determine the name of its supervertex. This is done by applying the optimal list ranking algorithm of (Cole and Vishkin, 1989) to the circular lists of blocks. It performs $O(m/d)$ operations in $O(\log n)$ time, where $d \geqslant (4 \log n \log^{(3)} n)^{1/2}$, the value of d at the end of the reduction procedure.

2. Rename each pseudo-growing supervertex with the name of its containing large supervertex (use the new name of its "first" block, computed in (1)). This takes $O(1)$ time and $O(m/d^2)$ operations.

3. Using the vertex table and the information computed in (2), for each vertex v, determine the large or growing supervertex containing v. This takes $O(\alpha)$ time and $O(n\alpha)$ operations.

4. Update the edge endpoints using the information computed in (3). This takes $O(1)$ time and $O(m)$ operations.

So computing the output graph takes a further $O(\log n)$ time and $O(m+n\alpha)$ operations.

We conclude

THEOREM 3.3. *There exists a CRCW PRAM algorithm that reduces the general connectivity problem to a connectivity problem for a graph with at most* $m/(\log n \log^{(3)} n)$ *vertices and at most* m *edges that performs* $O((m+n)\,\alpha(m,n))$ *operations in time* $O(\log n)$ *on* $(m+n)\,\alpha/\log n$ *processors. Hence there is a CRCW PRAM algorithm for connectivity with complexity* $O((m+n)\,\alpha(m,n))$ *operations and* $O(\log n)$ *time.*

APPENDIX 1: THE APPROXIMATE TASK SCHEDULING

We outline the solution to the approximate task scheduling of the Part I paper, or rather the following slight modification of this scheme: We are given s tasks, each of duration between 1 and t, and of total duration w; using p processors the tasks can be performed within the following times. All but possibly fp tasks, for some constant f given later, can be performed in $O(w/p+\log n/\log^{(2)} n)$ time. After the remaining (at most fp tasks) are properly redistributed among the processors they can be performed in an additional $O(t)$ time. For the problem of Section 3.3, this redistribution will take $O(\log n/\log^{(2)} n)$ time, where n is as defined there and this time is due to application of the CRCW parallel prefix sum algorithm. So altogether the s tasks will take $O(t+\log n/\log^{(2)} n+w/p)$ time using p processors and of course $O(pt+p\log n/\log^{(2)} n+w)$ operations.

This appendix is devoted to showing how to perform all but possibly fp tasks in $O(w/p)$ time based on the solution of the Part I paper. This bound applies even if w is not known in advance. (Note that the sequencing of the tasks is not predetermined. In fact, according to the sequencing of tasks, each individual task may have a varying length, but always in the range $[1, t]$; also, the sum of the lengths of the tasks is always bounded by w.)

Our solution requires repeated rapid rescheduling of the processors. The mechanism that provides each rescheduling is a *redistribution* procedure (it solves the *object redistribution* problem). The *object redistribution* problem is defined as follows. We are given r *objects*, partitioned among p *collections* of objects. We are also given one processor per collection. Loosely speaking, the problem is to redistribute the objects among the collections so that they are more evenly distributed. For a more precise description we

need some definitions. size_i, the *size* of collection i, $1 \leqslant i \leqslant p$, is the number of objects in collection i; note that $r = \sum_{i=1}^{p} \text{size}_i$. The *weight* of collection i is size_i^2, the square of its size; let $W = \sum_{i=1}^{p} \text{size}_i^2$ be the total weight of all the collections. Let $WMIN_r$ denote the minimum possible weight over all possible distributions of r objects among p collections; we note that, for $r \geqslant p$, $p \lfloor r/p \rfloor^2 \leqslant WMIN_r \leqslant p \lceil r/p \rceil^2 \leqslant 4WMIN_r$. When the value of r is clear from the context, we simply write $WMIN$. Let f and g be constants ($f \geqslant g = 298$, with $f = \max\{g, (8d^2)\}$, where d is the constant required for the expander graph needed for the redistribution problem; see the Part I paper). If W is upper bounded by either fp or $g\ WMIN$ then the collections are said to be *blanced* (i.e., either there are few objects present, or the objects are roughly evenly distributed).

The *object redistribution problem* is the following: Redistribute the objects (using the redistribution procedure) so that the following properties hold:

(1) The total weight is not increased.

(2) The maximum number of objects in any one collection does not increase.

(3) If the collections are unbalanced, the total weight of the collections is reduced by a multiplicative constant factor.

(4) The redistribution takes $O(1)$ time.

In the Part I paper we described the redistribution procedure and showed how to implement it in $O(1)$ time. Here we show how to use this procedure to solve the task scheduling problem; this is very similar to the solution of the slightly less general task scheduling problem solved in the Part I paper.

We solve the task scheduling problem as follows. When applying the redistribution procedure, we view the tasks as the objects. We define r to be the number of tasks at hand. Initially, we distribute s/p tasks to each collection. Each collection will never contain more than s/p tasks. A task is said to be *active* so long as it is not completed.

while ($> fp$ active tasks at hand)
  repeat $\log n / \log^{(2)} n$ times
    (i) For each processor, perform one real step on one active task in its collection, if any, in $O(1)$ time.
    (ii) Apply the redistribution procedure. This requires $O(1)$ time.

Remark. To check the while condition, we exploit the fact that the data structure for storing the active tasks stores, with each processor P, the number of active tasks belonging to P. The while condition is checked (i.e., the number of active tasks at hand is determined) by applying the

$O(\log n/\log^{(2)} n)$ time parallel prefix sum algorithm to sum the number of active tasks belonging to each processor.

The following lemma guarantees that this routine solves our problem.

LEMMA. *This routine performs all but at most fp of our tasks in* $O(w/p + \log n/\log^{(2)} n)$ *time on p processors.*

Proof. Consider a single iteration of the inner loop and consider the end of part (i) of that iteration. There are three possibilities.

Case 1. The collections are not balanced.

Case 2. The collections are balanced; so either

Case 2a. the total weight is bounded by g *WMIN* but not by *fp*, or

Case 2b. the total weight is bounded by *fp*.

In Case 1, part (ii) of the iteration reduces the total weight by a constant factor by property (3) of the redistribution problem. Since the initial weight is at most $p(\lceil s/p \rceil)^2$, this can happen only $O(\log s/p)$ times, say at most $h \log s/p$ times, for some constant h (for at that point the weight will have been reduced to *fp*; i.e. the collections are balanced).

We claim that in Case 2a at least $1/(16g) \cdot p$ of the collections are not empty following part (i) of the iteration. This is seen as follows. We first note that r, the number of objects present, is at least p for otherwise we would have $WMIN \leqslant p$. This would imply $g\,WMIN \leqslant fp$ (remember $f \geqslant g$), and therefore contradicts the assumption of Case 2a. Next, suppose x of the collections are non-empty. Then W, the total weight of all the collections, is at least $x \cdot \lfloor r/x \rfloor^2$; also $W \leqslant g\,WMIN$ and $WMIN \leqslant p \cdot \lceil r/p \rceil^2$. That is $x\lfloor r/x \rfloor^2 \leqslant gp\lceil r/p \rceil^2$, and since $r \geqslant p \geqslant x$, we have $r^2/4x \leqslant 4gr^2/p$, or $x \geqslant p/(16g)$.

For each non-empty collection, in part (i) of the iteration, one real step was performed on the task at hand. Thus, in Case 2a, at least $p/16g$ real steps were performed on the tasks in part (i) of this iteration. We conclude that there are at most $16gw/p$ instances of Case 2a.

In Case 2b, there are at most *fp* active tasks remaining.

We have shown that following $h \log(s/p) + 16gw/p$ iterations of the inner loop there can be at most *fp* active tasks remaining. Due to the stopping rule of the while loop these iterations come in bundles of $\log n/\log^{(2)} n$ and therefore take time $O(\log(s/p) + w/p + \log n/\log^{(2)} n)$. Since each task has length at least one, $s = O(w)$, and so $\log(s/p) = O(\log(w/p)) = O(w/p)$. Therefore these iterations take time $O(w/p + \log n/\log^{(2)} n)$. ∎

Data Structures. Instead of referring the reader to the Part I paper we refer rather to (Cole and Zajicek, 1990). There, a data structure was given

for storing the active tasks at each processor. This data structure supports the following constant time operations:

(i) A transfer of tasks from one processor to another processor as required by the redistribution procedure.

(ii) A deletion of a no longer active task.

APPENDIX 2: THE UNION–FIND PROBLEM

A special case of the UNION–FIND problem is solved. Suppose that initially n elements are provided on which parallel collections of UNION operations alternate with collections of FIND operations. Next suppose that at each point in time the sets can be divided into two categories, the *fixed* and the *changing* sets; it is guaranteed that no further UNIONs are performed on a fixed set, while further UNIONs may be performed on changing sets. A changing set may become fixed. Suppose further it is guaranteed that after a sequence of i (parallel) collections of UNION operations there are at most $cn/2^i$ changing sets at hand, for some constant c. This makes the problem considered here only an instance of the general UNION–FIND problem (see, e.g., (Aho *et al.*, 1974)).

We show how to maintain a UNION–FIND data structure that uses $O(n\alpha(m, n))$ operations and $O(\log n)$ time to perform all the UNIONs while performing each FIND in $O(\alpha(m, n))$ time and being able to perform any number of FINDs in parallel. Henceforth, we denote $\alpha(m, n)$ by α. The parameter m is unspecified; it is natural to choose it to be equal to the number of FIND operations, which is the choice for our application.

The sets are stored in a table T. For each element e, $T(e)$ is the name of the set currently containing e. We show how to maintain T, using a total of $O(n\alpha(m, n))$ operations and $O(\log n)$ time, so that, using a single processor, it takes $O(\alpha(m, n))$ time to determine $T(e)$. Each set of UNION operations is provided in the following form. The names of the new sets are provided in two arrays, one for the new fixed sets, the other for the new changing sets; also, each old changing set has a pointer to its new containing set.

We maintain the table T as a forest. Each changing set is the root of a tree of exactly $\alpha + 2$ levels. The leaves, level $\alpha + 1$ nodes, represent elements. The roots, level 0 nodes, represent the current sets. Internal nodes represent sets created at earlier phases of the processing. Each node, except a root, has a pointer to its parent in the forest. Initially, each tree is a chain of $\alpha + 2$ copies of the same element. Each fixed set is quite similar, except that it is the root of a tree with at most $\alpha + 2$ levels. It is helpful to consider the trees for fixed sets as missing a span $[1, i]$ of levels, for some i, $0 \leqslant i \leqslant \alpha$.

The fixed and changing sets are stored in one table, containing one array for each level $i \geqslant 1$, but two arrays for level 0; one of the arrays at level 0 contains the fixed sets, while the other contains the changing sets. We also record the number of nodes at each level, where by the "size" of level 0 we mean the number of its changing sets. Since the table is a forest, the sizes of the levels, going down towards the leaves, starting at level 1, are non-decreasing. A good intuitive guide is to view the sizes as being rapidly increasing (although this is not strictly correct all the time).

After every sequence of unions, the forest is updated as follows. We introduce new roots corresponding to new sets. Each new root is presented in exactly one array; either the fixed set array, or the changing set array. Each old changing set has a pointer to its new containing set. In effect, a forest of $\alpha + 3$ levels has been created. To perform the update we need to recreate a forest of $\alpha + 2$ levels; this is achieved by shortcutting over the old level 0 changing sets. The shortcutting is performed by the level 1 nodes in $O(1)$ time; the number of operations performed is proportional to the number of level 1 nodes. We call this process updating the level 0 nodes. In addition, the new fixed sets are placed at the end of the array containing the old fixed sets (this array is given size n to ensure it has sufficient space).

On occasion, we update the level $i \geqslant 1$ nodes (and consequently the parent pointers for the level $i+1$ nodes) as follows. (See Fig. 1.) This update requires the introduction of a new level $i-1$, a new level i, and changes the parent pointers at level $i+1$. We create a new instance of level $i-1$: it is a copy of the old level $i-1$. This will be the new level $i-1$ (if $i=1$, the copy of old level $i-1$ is a copy of the changing sets). The parent pointers in the old level $i-1$ are now changed to point to the corresponding nodes in the new level $i-1$. The old level $i-1$ will become the new level i. At this point we have created a forest of $\alpha+3$ levels; it remains to reduce it to $\alpha+2$ levels by shortcutting over the old level i. The shortcutting is performed by the level $i+1$ nodes in $O(1)$ time. The number of operations performed in updating level i is proportional to the size of level $i+1$ plus the size of level $i-1$; but this is proportional to the size of level $i+1$ (since the sizes of the levels, going towards the leaves, are non-

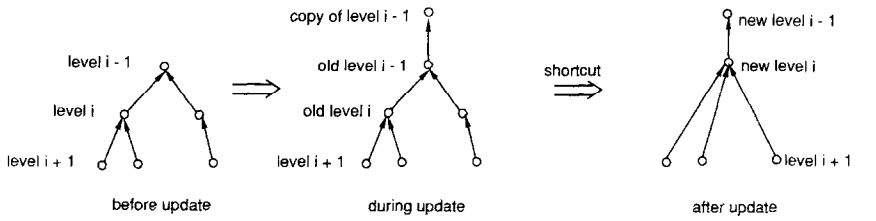

FIG. 1. An update of level 1.

decreasing). Again, since the sizes of the levels, going down the forest (towards the leaves), are non-decreasing, we can afford to update successively deeper levels of the forest less and less frequently.

At this point, it is helpful to recall the definition of Ackerman's function and its inverse:

$$A(1,j)=2^j \qquad \text{for} \quad j\geqslant 1$$
$$A(i,1)=A(i-1,2) \qquad \text{for} \quad i>1$$
$$A(i,j)=A(i-1,A(i,j-1)) \qquad \text{for} \quad i,j>1.$$

We define $A(i,0)=1$, for all i. (This last definition is merely for convenience. It is used in Lemma 1, below.)

We define $\alpha(m,n)$ to be the least i such that $A(i,\lceil m/n\rceil)>n$.

The procedure will need all the values of Ackerman's function $\leqslant n$. But these values are readily computed by a single processor in $O(\log n)$ time; this processor places these values in an array.

The followng rule indicates when updates are performed:

> Level i, for $1\leqslant i\leqslant\alpha$, is updated when the size of the $(i-1)$th level is first bounded by $n/[A(i,r)]^2$, for $r=0,1,\ldots$, in turn. In addition, we define the creation of each node at each level to be its first update.

We remark that the outermost squaring is present for technical reasons; for intuitive understanding it can be ignored.

LEMMA 1. *For $i\geqslant 1$, between two successive updates of level $i+1$, the second of which reduces level $i+1$ from size at most $n/[A(i+1,r)]^2$ to size at most $n/[A(i+1,r+1)]^2$, for some $r\geqslant 0$, level i is updated at most $A(i+1,r)$ times, for $r\geqslant 1$, and at most 2 times, for $r=0$ (for $r=0$, the first update to level $i+1$ is its creation).*

Proof. Between these two updates of level $i+1$, level i is reduced from size at most $s_1=n/[A(i,A(i+1,r-1))]^2=n/[A(i+1,r)]^2$, for $r>1$, from size $s_1=n/[A(i+1,1)]^2=n/[A(i,2)]^2$, for $r=1$, and from size $s_1=n=n/[A(i,0)]^2=n/[A(i+1,0)]^2$, for $r=0$, to size at most $s_2=n/[A(i,A(i+1,r))]^2=n/[A(i+1,r+1)]^2$, for $r\geqslant 1$, and to size at most $s_2=n/[A(i+1,1)]^2=n/[A(i,2)]^2$, for $r=0$. An update of level i reduces it from size at most $n/[A(i,j)]^2$ to size at most $n/[A(i,j+1)]^2$; thus reducing level i from size at most s_1 to size at most s_2 requires at most $A(i+1,r)-A(i+1,r-1)$ updates of level i, if $r>1$, at most $A(i+1,r)-2$ updates of level i, if $r=1$, and at most 2 updates of level i, if $r=0$. The lemma follows. ∎

LEMMA 2. *The updates to level* 0 *perform* $O(n)$ *operations in time* $O(\log n)$.

Proof. Each update requires $O(1)$ time and performs $O(|\text{level } 1|)$ operations. We start by considering the first $\lceil \log c \rceil$ updates, which reduce the bound on the number of changing sets to at most n. These require $O(\log cn)$ or $O(n)$ operations and time $O(1)$. Next, we consider the remaining updates to level 0. The result follows from the following claim: After the $2k$th such update level 1 has size at most $n/2^{2^k}$. For the $2k$th such update reduces the bound on the number of changing sets to at most $n/2^{2^k}$. Thus at this point, or earlier, level 1 is updated so as to reduce its size to at most $n/2^{2^k}$. ∎

LEMMA 3. *Level* α *is updated less than* m/n *times.*

Proof. Following $\lceil m/n \rceil$ updates level α would have size at most $n/[A(\alpha, \lceil m/n \rceil)]^2 < 1$. The lemma follows. ∎

LEMMA 4. *The total number of operations used to perform all the updates is* $O(n\alpha + m)$.

Proof. Each update to level α uses $O(n)$ operations; so, by Lemma 3, the updates to level α use $O(m)$ operations altogether. By Lemma 1, the updates to level i, $1 \leqslant i < \alpha$, for each $r \geqslant 1$, use $O((n/A(i+1, r)^2)\, A(i+1, r))$ operations and in total, $O(\sum_{r>0} n/A(i+1, r))$ operations. But $\sum_{i=1}^{\alpha-1} \sum_{r>0} n/A(i+1, r) = O(n)$. Again, by Lemma 1, the updates to level i, $1 \leqslant i < \alpha$, for $r = 0$, use $O(n)$ operations; over all the levels, this is $O(n\alpha)$ operations. Finally, the cost of the updates to level 0 is $O(n)$ by Lemma 2. ∎

Each update takes $O(1)$ time and there are $O(\log n)$ updates to all the levels (for the number of updates at level 0 is $O(\log n)$, at level 1 is $O(\log^{(2)} n)$, at level 2 is $O(\log^* n)$, and at all subsequent levels decreases by a factor of at least 2 from level to level; in fact it decreases by much larger factors). To know when to perform an update, we need to maintain a count of the number of nodes at each level of the forest. Level i acquires the size of level $i-1$ whenever level i is updated. Also recall that the nodes at each level are stored in an array; whenever the size of level i is reduced, the new level i nodes must be placed in contiguous locations in the array. But this is readily achieved, for a new level i is merely a copy of level $i-1$.

Thus, over the whole algorithm, maintaining the vertex table requires $O(n\alpha + m)$ operations and $O(\log n)$ time. The table uses $O(n\alpha)$ space. Clearly, each FIND takes $O(\alpha)$ time.

Remark. We can reduce the operation count for maintaining the vertex

table to $O(n+m)$ operations. It suffices to keep just one instance of "identical" levels, that is, levels of the same size. This also reduces the space requirement to $O(n+m)$ space. We leave it to the interested reader to verify the claimed complexity bound.

ACKNOWLEDGMENTS

We are very grateful to the referees for their help. One referee, Torben Hagerup, read the paper very carefully and thoroughly on three occasions; he discovered several non-trivial errors in the paper and made a substantial number of useful suggestions that led to a considerable improvement in the presentation of the paper. We also thank Jeanette Schmidt for translating the other referee's report from German into English.

RECEIVED April 22, 1987; FINAL MANUSCRIPT RECEIVED December 11, 1989

REFERENCES

AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. (1974), "The Design and Analysis of Computer Algorithms," Addison–Wesley, Reading, MA.

ATALLAH, M. J. (1984), Parallel strong orientation of an undirected graph, *Inform. Process. Lett.* **18**, 37–39.

AWERBUCH, B., ISRAELI, A., AND SHILOACH, Y. (1984), Finding Euler circuits in logarithmic parallel time, *in* "Proc. Sixteenth Annual ACM Symp. on Theory of Computing," pp. 249–257.

AWERBUCH, B., AND SHILOACH, Y. (1983), New connectivity and MSF algorithms for ultracomputer and PRAM, *in* "Proc. Int. Conf. on Parallel Processing," pp. 175–179.

ATALLAH M., AND VISHKIN, U. (1984), Finding Euler tours in parallel, *J. Comput. System Sci.* **29**, 330–337.

BRENT, R. P. (1974), The parallel evaluation of general arithmetic expressions, *J. Assoc. Comput. Mach.* **21** (2), 201–206.

COLE, R. (1987/1988), An optimally efficient selection algorithm, *Inform. Process. Lett.* **26**, 295–299.

CHIN, F. Y., LAM, J., AND CHEN, I. (1982), Efficient parallel algorithms for some graph problems, *Comm. ACM* **25**, 659–665.

COLE, R., AND VISHKIN, U. (1986a), Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control* **70**, 32–53.

COLE, R., AND VISHKIN, U. (1986b), Deterministic coin tossing and accelerating cascades: Micro and macro techniques for designing parallel algorithms, *in* "Proc. Eighteenth Annual ACM Symp. on Theory of Computing," pp. 206–219.

COLE, R., AND VISHKIN, U. (1986c), Approximate and exact parallel scheduling with applications to list, tree and graph problems, *in* "Proc. Twenty Seventh Annual Symp. on Foundations of Computer Science," pp. 478–491.

COLE, R., AND VISHKIN, U. (1989), Faster optimal parallel prefix sums and list ranking, *Inform. and Comput.* **81**, 334–352.

COLE, R., AND VISHKIN, U. (1987), "Approximate Parallel Scheduling. Part II: Applications to Logarithmic Time Optimal Parallel Graph Algorithms," TR 64/87, Dept. of Computer Science, Tel Aviv Univ., also, TR 291, Dept. of Computer Science, Courant Institute, New York University.

COLE, R., AND VISHKIN, U. (1988a), The accelerated centroid decomposition technique for optimal parallel tree evaluation in logarithmic time, *Algorithmica* **3**, 329–346.

COLE, R., AND VISHKIN, U. (1988b), Approximate parallel scheduling. I. The basic technique with applications to optimal parallel list ranking in logarithmic time, *SIAM J. Comput.* **17**, 1, 128–142.

COLE, R., AND ZAJICEK, O. (1990), An optimal parallel algorithm for building a data structure for planar point location, *J. Parallel Distrib. Comput.* **8**, 280–285.

EVEN, S. (1979), "Graph Algorithms," Computer Sci., Press, Rockville, MD.

GAZIT, H. (1986), An optimal randomized parallel algorithm for finding connected components in a graph, *in* "Proc. Twenty Seventh Annual Symp. on Foundations of Computer Science," pp. 492–501.

HAGERUP, T. (1988), Optimal parallel algorithms on planar graphs, *in* "Proc. 3rd AWOC," pp. 24–32, Lecture Notes in Computer Science, Springer-Verlag, Berlin/New York.

HAGERUP, T., CHROBAK, M., AND DIKS, K. (1989), Optimal parallel 5-coloring of planar graphs, *in* "Proc. 14th ICALP," pp. 304–313; *SIAM J. Comput.* **18**, 288–300.

HIRSCHBERG, D. S., CHANDRA, A. K., AND SARWATE, D. V. (1979), Computing connected components on parallel computers, *Comm. ACM* **22**, 461–464.

KOUBEK, V., AND KRSNAKOVA, J. (1987), Parallel algorithms for connected components in a graph, *in* "Fifth International Conference on Fundamentals of Computer Science," pp. 208–217, Lecture Notes in Computer Science, No. 199, Springer-Verlag, Berlin/New York.

KRUSKAL, C. P., RUDOLPH, L., AND SNIR, M. Efficient parallel algorithms for graph problems, *Algorithmica* **5**, 43–64.

LOVASZ, L. (1985), Computing ears and branchings in parallel *in* "Proc. Twenty Sixth Annual Symp. on Foundations of Computer Science," pp. 464–467.

MAON, Y., SCHIEBER, B., AND VISHKIN, U. (1986), Parallel ear decomposition search (EDS) and *st*-numbering in graphs, *Theoret. Comput. Sci.* **47**, 277–298.

SAVAGE, C., AND JA'JA', J. (1981), Fast, efficient parallel algorithms for some graph problems, *SIAM J. Comput.* **10**, 682–691.

SHILOACH, Y., AND VISHKIN, U. (1982), An $O(\log n)$ parallel connectivity algorithm, *J. Algorithms* **3**, 57–67.

SCHIEBER, B. AND VISHKIN, U. (1987), On finding lowest common ancestors: Simplification and parallelization, TR 63/87, Dept. of Computer Science, Tel Aviv University; *SIAM J. Comput.* **17**, 1253–1262.

TSIN, Y. H., AND CHIN, F. Y. (1984), Efficient parallel algorithms for a class of graph theoretic problems, *SIAM J. Comput.* **13**, 580–599.

TARJAN, R. E., AND VISHKIN, U. (1985), An efficient parallel biconnectivity algorithm, *SIAM J. Comput.* **14**, 862–874.

VISHKIN, U. (1983), Synchronous parallel computation—A survey, TR 71, Dept. of Computer science, Courant Institute, New York University.

VISHKIN, U. (1984), An optimal parallel connectivity algorithm, *Discrete Appl. Math.* **9**, 197–207.

VISHKIN, U. (1985), On efficient parallel strong orientation, *Inform. Process. Lett.* **20**, 235–240.

WYLLIE, J. C. (1979), The complexity of parallel computation, TR 79-387, Department of Computer Science, Cornell University, Ithaca, NY.