ON DIFFUSING COMPUTATIONS*

K. M. Chandy

J. Misra

TR-150 July 1980

* This work was supported by NSF Grant MCS-7925383.

Abstract

Dijkstra and Scholten developed the notion of diffusing computations to provide a very general framework for a large and important class of problems. In this paper we show how the results of diffusing computations can be compiled efficiently and applied to a variety of problems.

1. INTRODUCTION

The advent of multi-micro processor systems has spurred efforts in the development of parallel algorithms. Many parallel algorithms consist of sequences of computational phases, where each phase consists of several tasks which are executable in parallel. The effective implementation of such algorithms on message-passing systems requires distributed methods to determine that a phase has completed and to determine the results of one phase for use in the next. Dijkstra and Scholten [1] were the first to discover a distributed technique for detecting the termination of a distributed computation. Our work is based on theirs. We are concerned with methods to determine the results of a computational phase in a distributed manner.

2. THE DIJKSTRA-SCHOLTEN DIFFUSING COMPUTATION ALGORITHM

We next describe Dijkstra - Scholten algorithm. We modify the algorithm and notation slightly to make it more applicable to a class of graph problems that we are interested in. We are concerned with a special class of distributed computations called diffusing computations. A diffusing computation is carried out by a network of processes, one of which is a special process called the initiator. The network, N, is represented as a finite directed graph G where each vertex in G corresponds to a process. Process A can send messages directly to process B only if edge (A,B) exists in G. B is called a successor of A, and A is called a predecessor of B. A process can send messages

to any of its successors whenever it wants to. (This assumption is different from Hoare [3], where a process can send a message only if the receiver is ready to receive the message. Diffusing computations fall within Hoare's model if every process is always ready to receive messages from all of its predecessors.)

2.1 Process Behavior

A process is either _executing_ or _idle_. An executing process is one which can compute and may send messages without receiving any further messages. An idle process cannot compute nor send messages without receiving further messages. A process is always willing to receive messages. An idle process becomes executing on receipt of a message. An executing process may become idle after an arbitrary time period. A process may transit several times between idle and executing during a diffusing computation.

2.2 Network Behavior

Initially all processes, except the initiator are idle. The diffusing computation begins when the initiator sends a message to one or more of its successors which causes them to begin executing; these processes may in turn send messages to their successors (who may be idle or executing). Hence the name diffusing computation. The computation terminates when all processes except the initiator are idle.

2.3 Signalling Scheme

Dijkstra and Scholten have devised a signalling scheme to determine when a diffusing computation terminates. They assume that a process can send _signals_ to its predecessors (in addition to sending messages to

its successors). Signals are used only for termination detection; they have no effect on the computation.

A process may be either neutral or engaged depending upon its signalling status. A process's idle/executing status depends upon the process's computation, whereas its neutral/engaged status depends upon the process's signalling. An engaged process may be idle or executing, but a neutral process must be idle. The signalling scheme will enable the initiator to determine the instant at which all other processes become neutral (at which point all other processes must also be idle). We may think of the signalling scheme in the following fashion: every message sent by a process has two parts - the message content part and a "return receipt" containing the sender's identity. The receiver "tears off" the return receipt and eventually sends it back to the sender. The sender keeps a count of the number of receipts outstanding, i.e. yet to be returned to it. Upon receiving a return receipt, the count is decremented and the receipt discarded.

A process is defined to be (i) engaged if it has one or more return receipts that it has not yet sent back, and (ii) neutral if it has no receipts in its possession (either because it didn't receive any messages or because it returned all the receipts). The initiator is always engaged.

The first message that a neutral process p receives which causes it to go from neutral to engaged is special, and we call it p's engaging message and we call the associated receipt p's engaging receipt; the process that sent the message is called p's engager; the time that p became last engaged is called p's engaging time.

A process returns the engaging receipt only when (i) it has no other receipts in its possession and (ii) has received receipts for all the messages it has sent and (iii) is idle. Note that returning the engaging receipt causes the process to become neutral at which point it must also be idle from condition (iii). A process returns non-engaging receipts at arbitrary times; we may assume immediate return.

We can prove that the diffusing computation has terminated when the initiator has received all of its receipts. For a truly beautiful proof see [1]. We merely sketch the intuition behind a proof here. Suppose some process $p_{i(K)}$, other than the initiator is engaged. Then it must have an engager. Let it be $p_{i(K+1)}$. Then $p_{i(K+1)}$ must also be engaged and, if it is not the initiator, it must have received its engaging message earlier than $p_{i(K)}$ did, because only an engaged process can send messages. Continuing in this fashion, we build up a chain of engagers $p_{i(0)}, p_{i(1)}, \ldots$ from any process $p_{i(0)}$, and the engagement times decrease along the chain. Hence the chain cannot form a cycle and must therefore terminate at the initiator. Hence if any process is engaged, the initiator could not have received all its receipts.

3. RESULTS OF A DISTRIBUTED COMPUTATION

We consider the case where the result of a process p_i's computation is contained in some variable or set of variables S_i at the point of termination of the diffusing computation, and where the final result of the diffusing computation is some function of the S_i's. For example, consider the problem of knot detection, solved by Dijkstra [2].

3.1 Knots [2]

We say that vertex p is reachable from vertex q if there exists a

path from p to q. A vertex p is said to be in a knot if and only if for every vertex q reachable from p, p is reachable from q and there is at least one such vertex q. It is required to design a diffusing computation in a network N corresponding to a graph G to be initiated by an arbitrary process p, that will enable p to detect whether it is in a knot. We associate boolean variables reachable and canreach with each process q in the network; reachable (canreach) is set to true if any only if q is reachable from (canreach) p. Dijkstra has presented a diffusing computation to compute reachable and canreach. The initiator must determine not only when the diffusing computation ceases but also the result of the computation (which is true if and only if there are no processes with reachable = true and canreach = false).

We define the result of process i's computation to be a flag $subordinate_i$ which is set to true if and only reachable = true and canreach = false for process i. Thus, the knot algorithm is one example of a computation whose result is a function, $\neg(\bigvee_i subordinate_i)$, of the results of each process's computation. There are many other such examples. (See [5] for instance for implementation of discrete-event simulation as a diffusing computation.) We next present methods for the initiator to collate the results of all the process's computations.

4. DETERMINING RESULTS OF A DIFFUSING COMPUTATION

The obvious method for determining the result of such a diffusing computation is for the initiator to read the result S_i of process i, for all i, _after_ the initiator has detected that the diffusing computation

is over. However, this solution requires that the architecture be such that the initiator can read results from each process.

4.1 A Cancellation Scheme for Reporting Results

The idea in this method is that a process sends back its current result when it returns an engaging receipt. However, a process may become engaged and disengaged several times during the course of a diffusing computation, in which case the process will send back a sequence of results and every result except the very last one will be incorrect (because they will be out-of-date). Hence we must devise a scheme whereby a process may cancel the out-of-date results that it has sent. To allow such a cancellation scheme a process i maintains R_i, the result of the last computation that it sent back (i.e. when it last got disengaged); if no result was sent back then R_i is null. When a process returns its engaging receipt it must send back at least both its current result S_i and its last result R_i, with the objective of cancelling the last result R_i that it sent and replacing it by S_i. All the S_i's and R_i's that have been transmitted during the course of a diffusing computation will be held collectively by all the engaged processes at all times; at termination, the initiator will be the only engaged process and it can deduce the last result S_k of each process k from all the S_i's and R_i's it holds.

The ith process maintains two bags POS_i and NEG_i (for positive and negative) where NEG_i contains pairs of the form (R_k,k) denoting that process k has cancelled the out-of-date result R_k, and POS_i contains pairs (S_j,j) where process j has sent back S_j.

Every process returns a positive and negative bag with every receipt. When process i receives the two bags from its successor j it adds the positive bag to POS_i, and the negative bag to NEG_i. When process i disengages itself it sets S_i to the new (most current) result, adds (S_i, i) to POS_i; (R_i, i) to NEG_i if R_i is not null, and sends back POS_i and NEG_i with the engaging receipt. With receipts other than engaging receipts, two null bags are sent.

When process i receives an engaging message it sets $R_i \leftarrow S_i$, and POS_i and NEG_i to null.

Theorem 1 The computation maintains the following invariant.

$$\bigcup_{i \in E} POS_i = \bigcup_{i \in E} NEG_i + \bigcup_{k \in D} \{(S_k, k)\} \qquad (1)$$

and $$R_k = S_k \text{ if } k = D \cap E \qquad (2)$$

where all operations in (1) are bag operations and where E is the set of engaged processes, and D is the set of processes which have disengaged themselves at least once during the computation (but may be currently engaged or neutral).

Proof: Eqns (1) and (2) are vacuously true initially since E = {initiator}, D is null, $POS_{initiator} = NEG_{initiator}$ = null. The equations can be affected only when E or D changes, because S_i, R_i, POS_i and NEG_i (all i) can be changed only when E or D change. When process j gets engaged, eqn (1) is not altered since $POS_j = NEG_j$ = null at the point of engagement, and D and S_k, $k \in D$, are also not altered by a process getting engaged. Eqn (2) is maintained by process j setting $R_j \leftarrow S_j$ at the point

of engagement. Hence the invariants are maintained during an engagement.

Consider the sequence of events when process j gets disengaged. Process j's engager, say process i, must be engaged at this point. Let POS_i' and POS_i'' denote the values of POS_i before and after (respectively) the disengagement. Then

$$POS_i'' = POS_i' \cup POS_j' \cup \{(S_j'', j)\}$$

$$NEG_i'' = NEG_i' \cup NEG_j' \cup \{(R_j', j)\}$$

$$R_j' = S_j'$$

and hence the invariant holds.

Corollary When the initiator detects termination of the diffusing computation,

$$POS_{initiator} - NEG_{initiator} = \bigcup_{K \in D} \{(S_k, k)\}.$$

It follows from the corollary that the initiator can deduce S_k for every $k \in D$, from its local information (POS and NEG).

Note: The performance of the algorithm can be improved if we maintain POS_j and NEG_j disjoint for every $j \in E$, by cancelling occurrences of common elements.

4.2 Collating Results by a Second Diffusing Computation

The initiator starts a new diffusing computation to collate the S_i's. In this computation, process i maintains a set A_i of pairs (S_k,k), where S_k is the (final) result of process k in the original computation. Process i returns A_i with its very first engaging receipt and returns null with later engaging receipts and non-engaging receipts. When process i receives A_j from a successor j it sets $A_i \leftarrow A_i \cup A_j$. Initially A_i consists of the singleton (S_i,i).

When process i becomes engaged for the very first time it sends a message to each one of its successors. It ignores subsequent messages received (except to return receipts). It is self-evident that the second diffusing computation will terminate with the environment having a set containing a pair (S_k,k) for each process k that participated in the original diffusing computation.

5. APPLICATIONS

Dijkstra and Scholten developed diffusing computations to provide a very general framework for a large and important class of distributed programs. It is surprising that a single technique can be so general- the reader can best appreciate the generality for himself by running through several problems.

We list below a few of the problems amenable to this approach.

(1) Knot detection (discussed in section 4)

The application of the cancellation method results in a considerable simplification of Dijkstra's algorithm for knot detection.

Note that it is sufficient to count the number of vertices i for which subordinate_i = true, because if this count is 0 (zero) then (and only then) is the initiator of the diffusing computation in a knot. This fact allows the encoding of the bags POS and NEG as a simple count C. When process i disengates itself, it sends a number d with its engaging receipt where

$$d = \begin{cases} C_i + 1 & \text{if subordinate}_i \text{ has changed to true during the current engagement} \\ C_i - 1 & \text{if subordinate}_i \text{ has changed from true during the current engagement} \\ C_i & \text{if there is no change in subordinate}_i \text{ during the current engagement} \end{cases}$$

When a process j receives a number d from a successor it updates C_j to $C_j + d$. Clearly $C_{\text{initiator}}$ is a count of the number of processes for which subordinate_i = true at the termination of the computation.

(2) Maximal Strongly Connected Components

The problem is for a vertex V to determine whether it is in a non-trivial strongly connected component and, if it is, to determine the identity of all vertices in the component. The vertex V acts as the initiator and starts diffusing computations to label vertices reachable from V and vertices which can reach V. $S_j = \{j\}$ if j can be reached from v and j can reach v; $S_j = \{\ \}$ otherwise.

(3) Network Flow

We use the standard augmenting path approach. A diffusing computation phase is used to find an augmenting path to the sink and another phase is used to increase the flow corresponding to the augmenting path found (if any).

To demonstrate the elegance and utility of diffusing computations we consider one problem in detail.

(4) Shortest Path and Detection of Negative Cycles

We want to determine the shortest path from some vertex (say vertex 1) in a finite graph to all other vertices. If there is an edge from vertex i to vertex j, then it has finite weight W_{ij}. If there is a cycle of negative weight which is reachable from vertex 1, then all vertices reachable from that cycle are defined to have a minimum distance (from vertex 1) of $-\infty$. We construct a network of processes corresponding to the graph, and process 1 is the initiator. Each process has the weights of outgoing edges available to it. We assume (for purposes of brevity) that every vertex is reachable from vertex 1 because unreachable vertices will not play a part in the diffusing computation.

A straightforward attempt at constructing a diffusing computation to solve this problem may result in the following:

The ith process has a local variable d_i associated with it, which is the minimum distance to that process, determined at the current stage in the computation. Process i sends messages containing a number t_{ij} to process j where $t_{ij} = d_i + W_{ij}$. Thus the message from i to j

is the length of a path to j with i as its prefinal vertex. Initially $d_j = \infty$ for all $j \neq 1$, and $d_1 = 0$. Upon receiving a message t_{ij} from its predecessor i, process j, $j \neq 1$, takes the following actions.

If $d_j > t_{ij}$ then $d_j \leftarrow t_{ij}$ and send messages t_{jk} to all successors k, where $t_{jk} = d_j + W_{jk}$. (Note that if $d_j \leq t_{ij}$ no action is taken.)

This diffusing computation will not terminate because any process on a negative cycle will send and receive messages indefinitely often. Hence, we must modify our straightforward attempt, so that we detect negative cycles, and then force the diffusing computation to terminate. The Dijkstra-Scholten signalling scheme can be adapted for more than termination detection; we use it to detect negative cycles and to thus force the diffusing computation to terminate! The crucial difference between the Dijkstra-Scholten algorithm and ours is that a process may change engagers without disengaging itself.

The algorithm works in two diffusing computation phases. When the first phase ends we will have correctly determined the minimum distance only to all vertices which have a true minimum distance other than $-\infty$. In the second phase we identify vertices with a minimum distance of $-\infty$.

This algorithm for phase 1 differs in two essential aspects from the Dijkstra-Scholten scheme.

1. A process may change its engaging receipt during computation.
2. When the initiator detects "termination" of phase 1, there may still be some engaged processes in the network! For these processes the distance from the initiator is $-\infty$.

Sketch of the Algorithm (Phase 1):

<u>If</u> $d_j \leq t_{ij}$

<u>then</u> return the receipt for this message

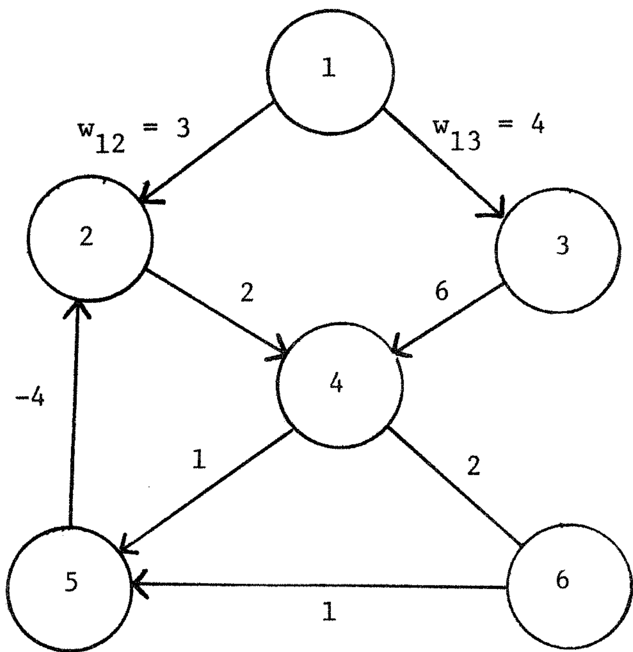
<u>else</u> $d_j \leftarrow t_{ij}$;

make <u>this</u> message the engaging message (the previous engaging receipt, if any, can now be returned);
send messages t_{jk} to all successors k where $t_{jk} \leftarrow d_j + W_{jk}$

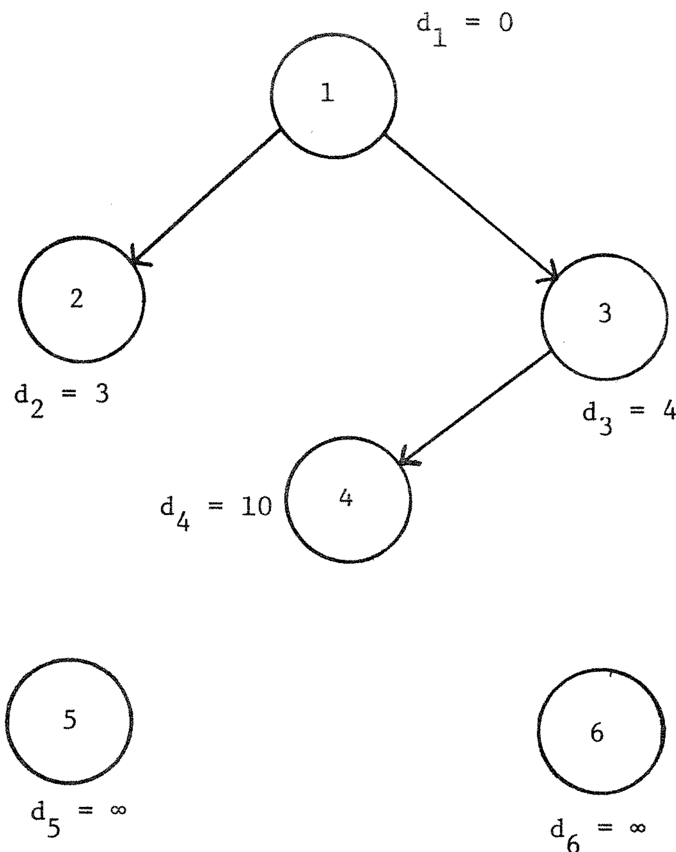
<u>end</u> phase 1

Example

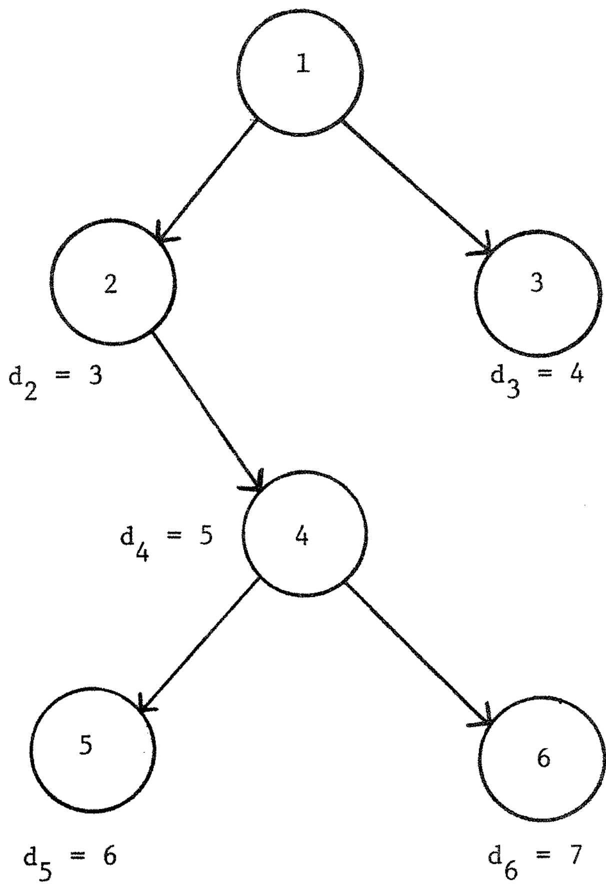
Consider figure 1(a). At some stage in the computation we may have the situation (figure 1(b)) in which processes 1, 2, 3 and 4 are engaged and 5 and 6 are neutral; 2's engager is 1 (represented by the edge (1,2)) 3's engager is 1 and 4's engager is 3. At this point process 4 may be sending messages 11 and 12 to processes 5 and 6 respectively, while process 2 is sending the message 5 to process 4. When process 4 receives message 5 it updates d_4 to 5, <u>changes its engager</u> to 2 and returns the receipt to process 3 and sends messages 6 and 7 to processes 5 and 6, respectively. The resulting situation is shown in figure 1(c). Now, since process 3 has no outstanding receipts, it returns its engaging receipt. Meanwhile, process 5 sends the message 2 to process 2. Process 2 <u>changes its engager</u> to process 5 and returns its receipt to process 1 resulting in figure 1(d). Now the initiator detects termination of, phase 1 even though processes 2, 4, 5 and 6 are engaged! (This is precisely the situation that Dijkstra and Scholten avoid! But the situation turns out to have some redeeming features.)



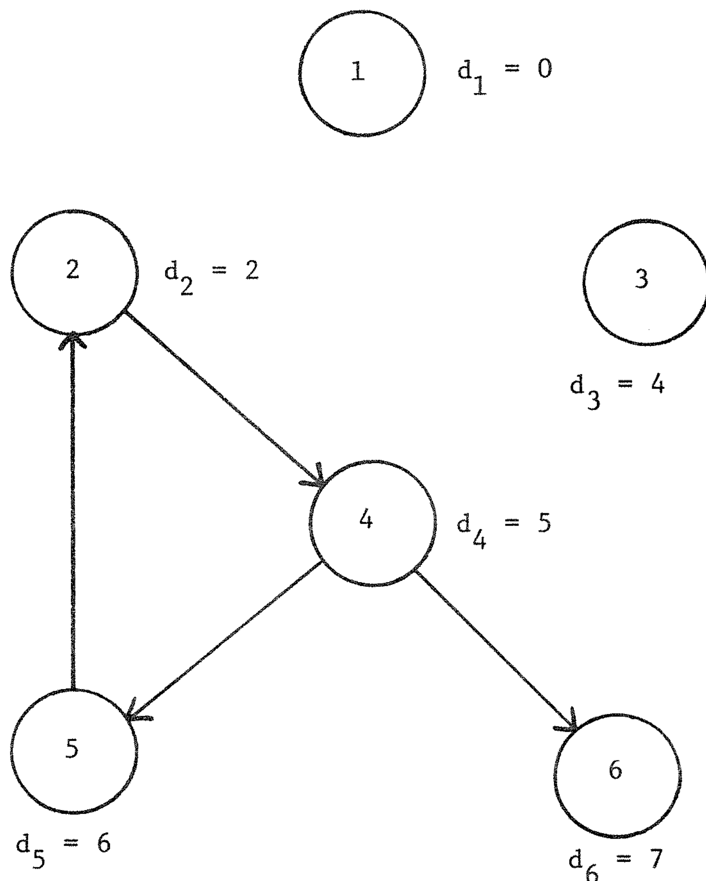
1(a)



1(b)



1(c)



1(d)

Process 1, the initiator, takes the following action on receiving a message t_{i1}:

<u>If</u> $0 \leq t_{i1}$

<u>then</u> return the receipt for this message

<u>else</u> halt phase 1 (since process 1 is in a negative cycle).

When the initiator receives all its outstanding receipts it halts phase 1 (if it has not already halted because it is on a negative cycle).

<u>Properties of the algorithm</u> (Proofs in appendix)

Define d_j^* to be the true minimum distance to process j.

<u>Theorem 2</u> Consider a process j for which $d_j^* \neq -\infty$. At the end of phase 1, (i) $d_j = d_j^*$ and (ii) process j is neutral.

<u>Theorem 3</u> If $d_j^* = -\infty$ then there exists a process i which can reach j and which is engaged at the end of phase 1. (Note that i could be the same as j).

<u>Corollary</u> At the end of phase 1 if (i) process j is engaged then $d_j^* = -\infty$ (ii) if every process on every path from 1 to j, (other than 1 itself) is neutral and if 1 is not on a negative cycle then $d_j = d_j^*$ (and $d_j^* \neq -\infty$).

<u>Theorem 4</u> Phase 1 terminates.

<u>Sketch of phase 2</u> The goal is to set $d_j = d_j^*$ for all processes j for which $d_j^* = -\infty$, and to then halt computation. This goal is accomplished as follows:

Set $d_j = -\infty$ for all processes reachable from a process (other than process 1) that remains engaged at the end of phase 1 and to force all processes in the network (other than process 1) to become neutral. If process 1 has detected that it is on a negative cycle, then all processes j have d_j set to $-\infty$ in phase 2.

Phase 2 employs two kinds of messages which show the end of phase 1: over? and over- . An over- message is sent by process j to all its successors if process j has determined that phase 1 is over and $d_j^* = -\infty$; an over- message orders the recipient to halt all phase 1 computation (if it has not done so already), set its d to $-\infty$ and propagate the over- message to its successors. An over? message is sent by process j to all its successors when it has determined that phase 1 is over, but has not determined whether $d_j^* = -\infty$. An over? message orders the recipient to halt all phase 1 computation; if the recipient was neutral at the end of phase 1 it sends over? messages to its successors, otherwise (if the recipient is engaged) it sends over- to its successors. Note that it is redundant to duplicate messages, and to send over? after over- . The computation starts by process 1 sending an over- message to all its successors if it has detected that it is on a negative cycle, and an over? message otherwise. It is self-evident that $d_j = d_j^*$ for all processes j at the end of phase 2 and that phase 2 will terminate.

The values of the minimum distances to all the processes can be determined by process 1 at the end of phase 2 by the methods of section 4.

6. CONCLUSIONS

Diffusing computation is a powerful method for solving a variety of graph problems, in parallel, on message-passing architectures. The Dijkstra-Scholten termination detection scheme is central to diffusing computation algorithms. We have shown (i) how to extend the Dijkstra-

Scholten algorithm to obtain the result of a diffusing computation and (ii) demonstrated the power of the Dijkstra-Scholten algorithm by using a modified version to solve the minimum distance problem and by indicating solutions to other graph problems.

APPENDIX

Proof of Theorem 2, part (i) Note that the algorithm keeps $d_j \geq d_j^*$ invariant because it is true initially, and if d_j is updated the new value of d_j is:

$$d_j = d_i + W_{ij} \geq d_i^* + W_{ij} \geq d_j^*$$

Assume the converse of part (i), i.e. $d_j \neq d_j^*$ and $d_j^* \neq -\infty$. Then $d_j > d_j^*$. Consider the case where process j is reachable from process 1, i.e. $d_j^* \neq \infty$ and consider a minimum distance path from process 1 to process j and let p_i be the first process on this path for which $d_i > d_i^*$. If p_i is process 1 then $d_i^* < 0$, and there is a negative cycle that includes process 1, and all processes j reachable from 1 have $d_j = -\infty$. Contradiction! Suppose p_i is a process other than process 1. Let p_k be the process preceding p_i on the shortest path from process 1 to process i. Clearly,

$$d_i^* = d_k^* + W_{ki}$$

First we note that d_k will be set to d_k^* at some time, since otherwise p_i would not be the first process on the path for which $d_i > d_i^*$; whenever d_k is reduced to d_k^*, process k sends a message $d_k^* + W_{ki}$ (which is d_i^*) to process i and hence process i must update d_i to d_i^*. Contradiction!

Proof of Theorem 2, part (ii) If process j is engaged, and there is no negative cycle on the shortest path from process 1 to process j, then j's engaging chain is a loop-free path which terminates at process 1.

Hence it is impossible for process 1 to receive all of its outstanding receipts while process j is engaged.

Proof of Theorem 3 If a process is on a negative cycle then after it is engaged, at all subsequent times, there exists at least one engaged process in the negative cycle, because at any time at least one process in the cycle must be having its d value reduced.

Assume the converse of the theorem; i.e. $d_j^* = \infty$ and every process which can reach j (except process 1) is neutral at the end of phase 1 Hence, any process k other than process 1, which is on a negative cycle and which can reach j, must never have been engaged. In particular, if j is on a negative cycle, then j has never been engaged.

Now consider the first process i with $d_i^* = -\infty$ on any simple path p from process 1 to process j. (Note: i may be 1 or j or any intermediate process.) If i = 1, the theorem is trivially true since process 1 is always engaged. Hence assume $i \neq 1$ and process i has never been engaged. Let k be i's predecessor on path p. If k ever got engaged then it will send a message to i, thus causing i to become engaged. Hence k has never been engaged either, which implies $d_k = +\infty$. Note that $d_k^* \neq +\infty$ since k is reachable from 1 along path p. Hence $d_k \neq d_k^*$ at the end of phase 1. Since i is the first process on path p with $d_i^* = -\infty$, we know that $d_k^* \neq -\infty$. Hence we have $d_k \neq d_k^*$ and $d_k^* \neq -\infty$ which contradicts theorem 2!

Proof of theorem 4 Assume the converse of the theorem, i.e. process 1 never receives all its outstanding receipts from its successors. Process 1 sends a message to a successor exactly once. Hence there must be some

successor of process 1, say process i_1 which never returns its engaging receipt to its engager (process 1). By a similar argument, there must be some process i_2, which got engaged by i_1 and never returned its engaging receipt to i_1. Continuing in this manner, there must be a cycle of processes within the sequence $p_1, p_{i_1}, p_{i_2}, \ldots$. But this is impossible since p_1 has no engager and all other processes in the sequence have exactly one engager. Contradiction!

REFERENCES

[1] Dijkstra, E. W. and C. S. Scholten, "Terminal Detection for Diffusing Computations, EWD 687a, Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

[2] Dijkstra, E. W., "In Reaction to Earnest Chang's Deadlock Detection," Plataanstraat 5, 5671 AL Nuenen, The Netherlands.

[3] Hoare, C. A. R., "Communicating Sequential Processes," CACM, Vol. 21, No. 8, August 1978.

[4] Ford, L. R. and D. R. Fulkerson, Flows in Networks, Princeton University Press, 1962.

[5] Chandy, K. M. and Jayadev Misra, "Diffusing Simulation: Parallel Simulation via Diffusion Computation," Technical Report, Computer Science Department, University of Texas, Austin, Texas 78712.