AN EFFICIENT PARALLEL TERMINATION DETECTION ALGORITHM

A. H. BAKER*, S. CRIVELLI†, AND E. R. JESSUP‡

Keywords: Termination detection; distributed termination; distributed computing

1. Introduction. Information local to any one processor is insufficient to monitor the overall progress of most distributed computations. When that is the case, a second distributed computation for detecting termination of the computation is necessary. In order to be a useful computational tool, the termination detection routine must operate concurrently with the main computation, adding minimal overhead, and it must promptly and correctly detect termination when it occurs [11]. In this paper, we present a new algorithm for detecting the termination of a parallel computation on distributed-memory MIMD computers that satisfies all of those criteria.

A variety of termination detection algorithms have been devised. See, for example, [1, 2, 5, 7, 8, 6, 10, 11, 12]. Of these, the algorithm presented by Sinha, Kale, and Ramkumar [11] (henceforth, the SKR algorithm) is unique in its ability to adapt to the load conditions of the system on which it runs, thereby minimizing the impact of termination detection on performance. Because their algorithm also detects termination quickly, we consider it to be the most efficient practical algorithm presently available. The termination detection algorithm presented here was developed for use in the PMESC programming library for distributed-memory MIMD computers [3, 4]. Like the SKR algorithm, our algorithm adapts to system loads and imposes little overhead. Also like the SKR algorithm, ours is tree-based, and it does not depend on any assumptions about the physical interconnection topology of the processors or the specifics of the distributed computation. In addition, our algorithm is easier to implement and requires only half as many tree traverses as does the SKR algorithm.

This paper is organized as follows. In section 2, we define our computational model. In section 3, we review the SKR algorithm. We introduce our new algorithm in section 4, and prove its correctness in section 5. We discuss its efficiency and present experimental results in section 6.

*Department of Applied Mathematics, University of Colorado, Boulder, CO 80309-0526, (allison.baker@colorado.edu). The work of this author was supported by the National Science Foundation under grant no. ACR-93-57812.

†Lawrence Berkeley Laboratory, Berkeley, CA (crivelli@global.lbl.gov). The work of this author was supported by the National Science Foundation under grant no. ACR-93-57812 and by the Department of Energy under grant no. DE-FG03-97ER25325.

‡Department of Computer Science, University of Colorado, Boulder, CO 80309-0430 (jessup@cs.colorado.edu). The work of this author was supported by the National Science Foundation under grant no. ACR-93-57812 and by the Department of Energy under grant no. DE-FG03-97ER25325.

2. Computational Model. We assume a distributed-memory computer with each processor assigned a different portion of the original problem. Our model of execution is distributed, asynchronous, and dynamic with one process per processor. The processors share computational tasks via message passing.

We distinguish between two concurrent computations: main and termination. The main computation solves the original problem. The termination computation detects the completion of the main computation on all processors. The main computation is further characterized as follows:

- Processors are either *active* or *passive*. A processor is active when it is working to complete its assigned computational tasks. Otherwise, it is passive.
- Only active processors can send tasks to other processors.
- Both active and passive processors can receive tasks from other processors.
- An active processor becomes passive only when it finishes its assigned computational tasks.
- A passive processor becomes active only if it receives tasks from another processor.

Messages used by the termination algorithm are called *control* messages to distinguish them from the *primary* messages used by the main computation to transfer tasks or data. Primary messages may change the status of the receiving processor from passive to active. Control messages cannot change the status of any processor.

The main computation has *terminated* if and only if

- All processors are passive and
- There are no primary messages left in transit.

A termination detection algorithm must detect both termination conditions.

3. The SKR Algorithm. The SKR algorithm [11] is a wave-type algorithm that requires the use of a virtual spanning tree for termination messages, but its description is not dependent on the properties of the physical underlying network topology. The algorithm is asymmetric since a root processor is identified. Neither FIFO communication channels nor synchronous communication are required.

The SKR algorithm does not interfere with the main computation. Processors do not handle the control messages until they are passive, and, in a busy system, few control messages are generated. As a result, the SKR algorithm is very efficient for the computational model described in the previous section. Further discussion of efficiency follows in section 6.

The SKR algorithm detects termination by counting the number of primary messages sent and received. It is organized in two phases and uses three types of termination messages (initialization, idle, and activity). Communication follows the links of a virtual spanning tree. In the first phase of the algorithm, when a leaf processor becomes passive, it sends a message to its parent in the tree containing the numbers of primary messages it has sent and received at that time. Once an internal processor becomes passive, it waits for idle messages from all of its children, adds the received counts to its own count, and forwards the result as an idle message to its parent.

The total counts in any processor represent the number of primary messages sent and received so far by all processors in the subtree rooted at that processor. Thus, if the total counts received by the root of the spanning tree are not equal, termination cannot have occurred, and the root broadcasts a signal to the leaves to reinitiate the first phase. However, if the root's totals are equal, termination may have occurred. Thus, the root broadcasts a signal to the leaves to initiate the second phase. The second phase detects whether any messages are still in transit even though the totals at the root match.

In the second phase, the processors, beginning at the leaves, send activity messages containing their updated counts of primary messages sent and received up the tree. Activity messages are combined in the same way as idle messages are in the first phase. When the root has received activity messages from all of its children, it compares the old totals of messages sent and received with the new ones. If these values are the same, there has been no activity in the system. In this case, the root reports termination. Otherwise, it restarts phase one with a broadcast to the leaves.

Because the progress of this termination detection algorithm is controlled by the root, it requires the broadcasting of a message from the root to the other processors to initiate each of the phases. The SKR algorithm therefore takes four traverses of the spanning tree to detect termination. In the next section, we present a new termination detection algorithm that is in the same class as the SKR algorithm. The new algorithm, however, takes at most two traverses of the spanning tree to detect termination after it occurs.

4. The New Algorithm. The new termination detection algorithm runs on each of the processors in the distributed-memory computer. All communication in the termination procedure takes place up and down a virtual spanning tree of processors. Pseudo-code for the termination algorithm is given in Figure 4.1.

The algorithm requires one phase and only two types of control messages: down and up. Down messages traverse the tree downwards carrying only the number of the termination sweep (called a sweep_#) they belong to. Up messages traverse the tree upwards and prompt each processor to evaluate the number of primary messages it has sent minus the number of primary messages it has received. Up messages carry

S	`local count of primary messages sent`
R	`local count of primary messages received`
T	`local count of sends minus receives`
accum_count	`accumulated count of sends minus receives from children`
children	`number of children that have sent termination messages`
all_children	`total number of children that belong to a processor`
sweep_#	`number of the termination sweep`
last_msg_sweep_#	`greatest sweep_# attached to work received by a processor`

```
Downward traverse
    if (not root) {
        check for down message containing a sweep_# from parent; }
    if (not leaf) {
        send down message containing sweep_# to children; }
    else { \* processor is a leaf *\
        initiate Upward traverse:
        T = S - R;
        send up message containing T and sweep_# (from down message) to parent; }
    return to main computation;

Upward traverse
    if (not leaf) {
        check for up messages from children containing T and sweep_#;
        if (message received) {
            accum_count = T (received) + accum_count;
            children++; }
        if (children = all_children) {
            if (last_msg_sweep_# > local sweep_#) {
                T = INFINITY; }
            else {
                T = S - R; }
            accum_count = T + accum_count;
            if (not root) {
                send up message containing accum_count and sweep_# to parent;
                set local sweep_# = sweep_# in up message just sent; }
            else { \* root processor *\
                if (accum_count = 0) {
                    broadcast termination; }
                else {
                    go back to Downward traverse; } } } }
    return to main computation
```

FIG. 4.1. *Pseudo-code for the termination detection algorithm.*

a sweep_# as well as an accumulated count of sends minus receives from processors below it in the tree.

The downward traverse starts at the root. When the root processor becomes passive, it sends a down message to its children. The sweep_# included in the down message is assigned by the root. The first time the root initiates a downward traverse it assigns the down messages sweep_# = 1. The root increments the sweep_# by one for each subsequent downward traverse that it initiates so that each termination sweep has a unique sweep_#. When any processor other than the root becomes passive, it checks to see whether it has received a down message from its parent. If so and if the processor is not a leaf, it forwards the sweep_# just received to its children as a down message. If the processor is a leaf, it evaluates the difference between the number of primary messages it has sent and received until that time. The leaf processor then sends its message count and the sweep_# contained in the down message as an up message to its parent in the tree. In this way, the leaf processors initiate the upward traverse.

In addition to being attached to the control messages, a termination sweep number is attached to all primary messages. Each processor keeps a local quantity called sweep_#, which is initialized to zero. A processor attaches its local sweep_# value to all primary messages it sends. A second quantity kept by each processor is last_msg_sweep_# (also initialized to zero). Each time a processor receives a primary message from another processor, it checks to see if the sweep_# included with the primary message is greater that its last_msg_sweep_#. If so, the processor updates its last_msg_sweep_# to equal the sweep_# included in the primary message. In this way, the local last_msg_sweep_# kept by each processor reflects the greatest sweep_# value attached to any primary message it has received so far.

Although every processor must be passive at some point during the downward traverse, completion of the downward traverse does not signal termination of the main computation: a processor may still receive a primary message and resume work after sending its down or up messages. The purpose of the upward traverse is to detect any primary messages that were in transit during the downward traverse. Note that the upward traverse can begin at a leaf processor before the downward traverse has reached all processors.

An internal processor collects up messages containing local message counts from all of its children. It forms an accumulated message count equal to the sum of its children's local message counts. When the internal processor has received up messages from all of its children, it evaluates its own message count. It then checks to see if it previously received any primary messages from processors that had completed the current termination sweep (i.e., sent both down and up messages in that sweep). Such a message was received if the checking processor has its last_msg_sweep_# greater than its sweep_#, requiring it to reset its message count to INFINITY. The processor then adds its own local message count to the accumulated message count and sends the result up to its parent in the tree. Immediately after an internal or leaf processor sends an up message, it updates its local sweep_# to equal the value of the sweep_#

in the up message just sent. In this way, the local sweep_# kept by each processor reflects the last termination sweep it completed. Note that an internal processor must be passive before it checks for an up message.

When the root receives up messages from all of its children, it determines whether or not termination has occurred. The root forms an accumulated message count (including its own local message count) in the same manner as the internal processors. If the value of its accumulated count is zero, then all the processors have finished, and the root broadcasts a termination order. Otherwise, termination has not occurred and the main computation continues on all processors until the root becomes passive and reinitiates the termination procedure.

The complexity of this algorithm is half that of the SKR algorithm presented in [11] because this algorithm requires only two traverses of the tree to detect termination. We do note, however, that for an application with fully synchronous communication, the second phase (requiring 2 traverses) of the SKR algorithm is not required. In both algorithms, one additional traverse is required for the root to broadcast the termination signal to all of the other processors after termination is detected.

A termination algorithm is fault tolerant if it correctly detects termination despite faults that occur at a processor or in the network during the computation. We note that neither our new algorithm nor the SKR algorithm is fault tolerant.

5. Proof of Correctness of the New Algorithm. In this section, we denote the set of processors in the distributed-memory computer by $\mathcal{P}$. Let s_m denote the sending of primary message m, and let r_m denote the receiving of the message. We identify the following sets: $S_{p_i} = \{s_m,\ \text{such that } m \text{ is a message sent by } p_i\}$, $S = \{S_{p_i},\ p_i \in \mathcal{P}\}$, $R_{p_i} = \{r_m,\ \text{such that } m \text{ is a message received by } p_i\}$, and $R = \{R_{p_i},\ p_i \in \mathcal{P}\}$. Let $\#S$ and $\#R$ be the cardinalities of S and R and $T = \#S - \#R$.

THEOREM 1. *The termination detection algorithm does not detect false termination.*

PROOF. The proof is by contradiction. Assume that the termination procedure detects false termination. In that case, the procedure has reached the root of the tree with $T = 0$ but termination has not yet occurred. Assume that the termination sweep that found $T = 0$ had sweep_# = b. According to the definition of termination given in section 2, if termination has not occurred then at least one of the following is true: ($\exists\ p_i \in \mathcal{P}$ such that p_i is active) or ($\exists$ a message m such that m is in transit).

First assume that p_i is active after termination is detected. Then p_i changed from passive to active after it was visited during the upward traverse, and its local sweep_# = b. A passive processor can become active only if it receives a primary message from another processor. So p_i necessarily received a primary message γ from another active processor p_j. Without loss of generality, we can assume that p_j had not yet been visited by the upward traverse when it sent the message although it may or may not have been visited by the downward traverse. (Any set of processors that are busy below the wavefront of the upward traverse must ultimately have been restarted by a message from a busy processor above the wavefront.) Since p_i is now active it could either (a) send no primary messages or (b) send a primary messsage to another processor. In case (a), $\#S \neq \#R$ since the sending of γ by p_j was counted in the termination sweep but the receiving of γ was not (γ was received by p_i after the up travese). Therefore, $T \neq 0$. In case (b), p_i sends a primary message β containing sweep_$\#$ = b to another processor p_k. Processor p_k must be in either of the following 2 stages of the termination procedure:

1. p_k has *not* been visited by the up traverse.
2. p_k has been visited the up traverse (like p_i)

If item 1 is true, then p_k's sweep_$\#$ < b since it has not completed the upward traverse for this sweep, but its last_msg_sweep_$\#$ = b because it received β. Therefore, p_k's up message will contain $T = INFINITY$, and T cannot be 0 upon completion. If item 2 is true, the same situation exists for p_k that was just described for p_i. Processor p_k is now active and can either (a) send no primary messages or (b) send a primary messsage to another processor. In case (a), $T \neq 0$ again because the sending of γ was counted, but the receiving of γ and the sending and receiving of β were not. Considering case (b) again, note that at this point messages could be sent any number of times to processors that have already completed the sweep, and $T \neq 0$ upon completion of the sweep. In this case, the only way to arrive at $\#S = \#R$ would be to have another receive counted by a processor that has *not* completed the upward traverse. Assume a primary message is sent by p_k (or any other active processor that has completed the current sweep) to any processor p_e that has *not* completed the upward traverse. Processor p_e will set $T = INFINITY$ when it sends its up message since its last_msg_sweep_$\#$ will be greater than its sweep_$\#$. Thus, our first assumption is false.

Now assume that there exists at least one message in transit at the completion of the termination sweep. Since $T = 0$, then by the definitions of S and R there exist at least two messages in transit, m_1 and m_2, such that the sending of m_1 and the receiving of m_2 have been counted but the receiving of m_1 and the sending of m_2 have not. Thus, there are messages m_1 and m_2 such that ($s_{m_2} \ni S$ and $r_{m_2} \in R$) and ($s_{m_1} \in S$ and $r_{m_1} \ni R$). Let p_k be the processor that sends m_2 and p_j be the processor that receives m_2. Because s_{m_2} is not in S then m_2 was sent after p_k was visited in the upward traverse. Note that m_2 had to contain sweep_# = b since the sending node p_k had already been visited by the upward traverse, making p_j's last_msg_sweep_# $\geq$ b. Since r_{m_2} is in R then it occurred *before* processor p_j was visited by the upward traverse. Therefore, when the upward traverse visits processor p_j, p_j will have last_msg_sweep_# $\geq$ b and sweep_# $<$ b, so $T = INFINITY$. Thus, our second assumption is also false. $\diamond$

We have proved that when termination is detected the system has terminated.

6. Efficiency of the New Algorithm. The efficiency of a termination detection algorithm has traditionally been equated with the number of control messages the algorithm creates. In particular, an algorithm is classified as "message optimal" if it uses a number of control messages on the order of the number of primary messages sent by the main computation [6]. We note, however, that the number of control messages is not a meaningful metric when control messages are handled only by passive processors. In this case, a better measure of efficiency is the overall impact the termination detection computation has on the performance of the main computation. Thus, an efficient termination detection algorithm must not interfere with the main computation. Furthermore, it must be able to detect both of the termination conditions identified in section 2 and communicate them to all of the processors quickly [11]. The SKR algorithm satisfies both of these efficiency requirements and typically requires far fewer control messages than the message optimality bound would dictate [11]. In this section, we present experimental results to demonstrate that our algorithm is likewise efficient in practice.

We examine a problem contrived to isolate the impact of our termination detection routine in a roughly worst case scenario. The experiments were carried out on an IBM SP. Test programs use Fortran (mpxlf -O), and MPI [9] is used for message passing.

In this problem, all processors begin by completing one short-lived task. When the root completes its task, it sends work to a given leaf processor and becomes passive. When that leaf processor completes its task, it sends work to the root and becomes passive itself. This cycle repeats five times. In the static implementation, the root and one leaf processor terminate computation after the fifth cycle. The remaining processors terminate computation upon completing the preliminary task. Thus, no termination detection routine is actually needed. For purposes of comparison, however, we have written a dynamic implementation that follows the same steps but also employs our termination detection algorithm. The expected impact of termination detection on this problem is high as the runtime is dominated by the communication cycles between root and leaf, and the root processor initiates a new termination

Number of Processors	Dynamic	Static
2	3.109 ± 0.035	3.107 ± 0.029
4	3.091 ± 0.033	3.089 ± 0.023
8	3.113 ± 0.067	3.122 ± 0.065
16	3.118 ± 0.034	3.116 ± 0.027
32	3.094 ± 0.036	3.092 ± 0.030
64	3.098 ± 0.030	3.094 ± 0.031

TABLE 6.1

Total execution times (in seconds) with and without termination detection on an IBM SP. Times reported are averages and standard deviations of times for 1000 runs.

detection procedure every time it changes from active to passive. Nonetheless, the results reported in Table 6.1 show that adding termination detection has little effect on the total runtime, even in this worse case. The times reported in the table are the averages and standard deviations of the times recorded for 1000 runs of the program. Differences in system load, numbers of simultaneous users, etc. cause fluctuations in experimental timings on the SP. These results demonstrate that the termination detection routine imposes an overhead smaller than the experimental uncertainty.

Acknowledgments. Experiments were run on the IBM SP at Argonne National Laboratory.

REFERENCES

[1] S. Chandrasekaran and S. Venkatesan. A message-optimal algorithm for distributed termination detection. *Journal of Parallel and Distributed Computing*, 8:245–252, 1990.

[2] M. Chandy and J. Misra. An example of stepwise refinement of distributed programs: Quiescence detection. *ACM Trans. on Programming Languages and Systems*, 8(3):326–343, 1986.

[3] S. Crivelli. *A programming paradigm and library for distributed-memory computers*. PhD thesis, Dept. of Computer Science, University of Colorado at Boulder, 1995.

[4] S. Crivelli and E.R. Jessup. The PMESC programming library for distributed-memory MIMD computers. *Journal of Parallel and Distributed Computing*, 57:295–321, 1999.

[5] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.

[6] J. Matocha and T. Camp. A taxonomy of distributed termination detection algorithms. *The Journal of Systems and Software*, 43:207–221, 1998.

[7] F. Mattern. Global quiescence detection based on credit distribution and recovery. *Information Processing Letters*, 30:195–200, 1989.

[8] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[9] Message Passing Interface Forum. MPI2: A message passing interface standard. *High Performance Computing Applications*, 12(1–2):1–299, 1998.

[10] S.P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 17:43–46, 1983.

[11] A. Sinha, L.V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report Internal Report 93-11, Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign, 1993.

[12] G. Tel and F. Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Trans. on Programming Languages and Systems*, 15(1):1–35, 1993.