Experience with a New Distributed Termination Detection Algorithm

Friedemann Mattern

Department of Computer Science, SFB124, University of Kaiserslautern,
P.O. Box 3049, D 6750 Kaiserslautern, Federal Republic of Germany

Abstract

A termination detection algorithm for a general model of distributed computations where processes communicate over asynchronous non-FIFO channels is presented. It has O(mn) message complexity if the control network is a ring, a (spanning) tree, or a general undirected graph and O(m) message complexity on star networks and complete networks. Several variants of the basic principle are discussed, one of which is a symmetric version where any process can start the algorithm independently from the other processes. Preliminary experimental results show that far less control messages than indicated by the worst case behavior are usually generated. In a distributed puzzle-solving system used as a test application only about $\sqrt{m}$ control messages have been counted. The constraint based puzzle-solving method is explained and several test cases are reported.

1. The distributed termination problem

Detecting when a distributed computation has terminated is an interesting and non-trivial problem. Let us consider a distributed system of n processes, where each process is either *active* (still computing and possibly sending messages to other processes) or *passive*. A passive process does not send messages and can only become active again when receiving a message. If at some instant in time all processes are passive and no message is in transit, no further computation is possible and the system is *terminated*. Since a priori no process has complete knowledge of the global state, no process can, without further action, decide whether the system has terminated. The *distributed termination problem* consists in devising an algorithm which can be superimposed on the original computation and allowing processes to detect the global termination condition by means of additional *control communication*.

A surprising variety of termination detection algorithms with varying assumptions about the underlying model of distributed computation has been proposed in recent years (cf. among others [FRA80], [FRR82], [TOP84], [SHF86]); and an overview of their different properties can be found in [BMR85] and [TAL86]. Whereas most solutions are based on a synchronous model of communication (notably CSP) where messages cannot be in transit if all processes are passive, we present and discuss a solution for a more general model where message passing is *asynchronous* and messages are not necessarily delivered in the order they were sent. Algorithms for such a model have been previously described in [DIS80], [KUM85], [LAI86], [MAT87a], and [MAT87b]. Compared to these solutions the

This work was supported by the Deutsche Forschungsgemeinschaft (DFG) as part of the INCAS project (SFB124-D1). Electronic mail address: mattern@incas.UUCP or mattern%uklirb.uucp@Germany.csnet

vector method described here has a number of interesting properties, among others it has *bounded message complexity* and does not require the basic messages (i.e. the messages of the underlying computation) to be time-stamped. First implementations show that it typically uses far less control messages than can be expected from the worst case behavior.

The rest of the paper is organized as follows: After presenting the principles of the vector termination detection algorithm in Section 2, Section 3 describes a distributed puzzle-solving system used as a test application for the empirical evaluation of the termination detection algorithm. Its implementation is outlined in Section 5 preceded by some aspects of the distributed high-level programming language CSSA in Section 4. In Section 6 the results of the experiments concerning the message complexity are displayed. Several important variants of the algorithm for various control topologies are discussed in Section 7. Finally, we present our conclusions in Section 8.

2. The principle of the vector algorithm

We consider systems with a fixed number of processes P_j ($1 \le j \le n$) connected by *unidirectional channels* represented by a *strongly connected* (directed) graph. For simplicity of exposition of the vector algorithm we assume that *control messages* travel along an unidirectional *control ring* connecting all processes independently of the channels for the *basic messages* of the underlying computation. This control topology will be generalized to arbitrary sequential (and parallel) traversal schemes in Section 7.

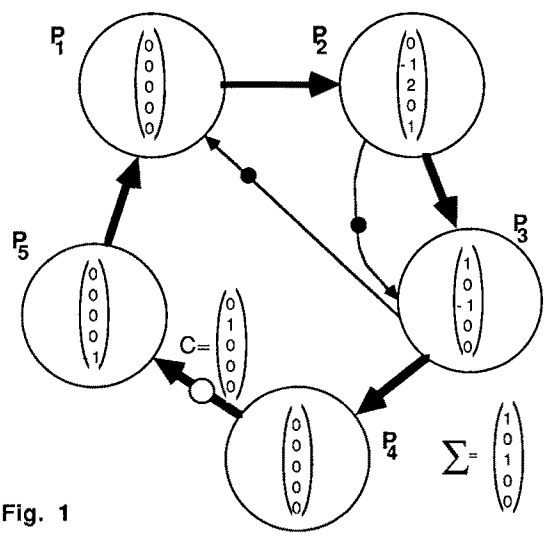

Fig. 1

Every process P_j has a local vector variable V[1:n] of length n, initialized to the null vector [0,...,0]. Whenever P_j sends a basic message to another process P_i, it increments V[i] by one, and whenever it receives a basic message from any other process it decrements V[j] by one. After the start of the basic computation a *control vector C* begins to circle around the ring, accumulating the local values of V and resetting them to [0,...,0] as it passes by. The circulating vector signals global termination when it becomes the null vector, has made at least one round, and has found all processes passive during the last round.

Figure 1 shows a snapshot of a distributed computation with 5 processes, the history of the computation from a global point of view is depicted in Figure 2 (t_2 is the time instant of the situation depicted in Figure 1). The duration of the active phases is irrelevant if a process hands over the control vector only when it is passive. The 'resting' of the circulating vector at a process is signaled by a local flag HAVE_VECTOR. Initially it is set to false, when the circulating vector is received by a process it is set to true and when it leaves the process it is reset. V[j] could be initialized to -1 in every process P_j - with all other components of V remaining 0 - in order to guarantee that at least one round is completed; the circulating vector then starts with [1,...,1]. However, this initialization trick is not used in Figure 2.

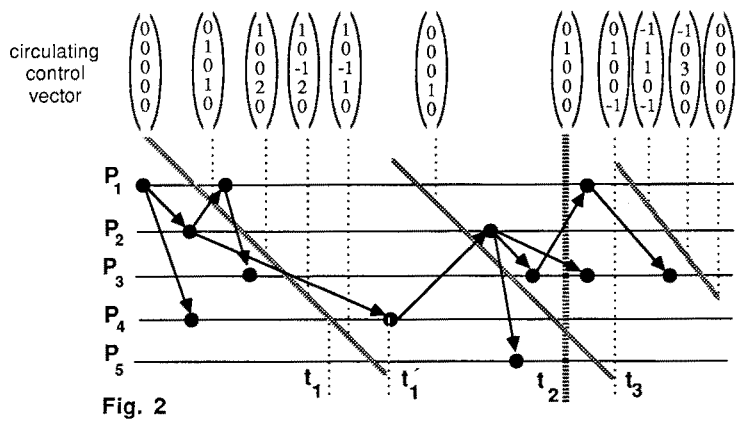

Fig. 2

The local vector V indicates how many messages have been received and sent since the last visit of the control vector. At any global time instant, the sum of the k-th components of all n local vectors V (including the circulating control vector C when it is in transit between two processes) equals the number of basic messages currently on their way to P_k. By virtue of the algorithm, this is a *global invariant*. For the situation depicted in Figure 1 the sum vector Σ indicates that a message to process P_1 and another message to P_3 are yet to be received.

The analysis of Figure 2 shows that at time instant t_1, when the control vector C visits P_4, P_4 can deduce from C[4]+V[4] = 2-1 = 1 that there is currently a message on its way to P_4. Instead of propagating the control vector as soon as it becomes passive, P_4 could keep the control vector until C[4]+V[4] becomes 0 at time instant t_1'. By generalizing this idea it can be guaranteed that at least one basic message is received in every round of the control vector (with the possible exception of the initial round). From this property follows inductively that the number of control messages is *bounded by n(m+1)*, where m denotes the number of basic messages (see [MAT87a] for a more detailed proof).

Figure 2 also shows that counting the total number of basic messages received and sent by simple *scalar counters* instead of counting them individually by vectors leads to false results. The value of a simple counter is identical to the sum of all components of a vector. At t_3 the counter would be 0 although the computation is not yet finished. This problem is analyzed in more detail in [KUM85] and [MAT87a]. Generally it can easily be proven by algebraic means that no coding of the vector by a linear combination $L = \Sigma\, a_k V[k]$ is sufficient.

The main factors of the vector algorithm are rather short; the counting of basic messages and initializations are as described in the text while every process P_j behaves as follows:

a) When receiving the control vector C:

$V \leftarrow V+C$;
HAVE_VECTOR $\leftarrow$ **true** ;

b) Whenever the control vector has been received or P_j becomes passive, the guard (i.e. the boolean expression of the first line) of the following 'demon' is evaluated:

```
if PASSIVE and HAVE_VECTOR and V[j] ≤ 0 then
    if V = [0,...,0] then TERMINATED ← true ;
    else
        HAVE_VECTOR ← false ;
        send V to P(j mod n)+1 ; /* forward the control vector V=C */
        V ← [0,...,0] ;
    fi
fi
```

Evaluation of the guard and execution of the demon must be performed atomically, i.e. the code has to be run to completion before the next basic message or control message is accepted.

The algorithm is *started* by one (and only one) process which gets the privilege to generate the control vector by setting HAVE_VECTOR to true and evaluating the guard. The local variable TERMINATED should be initialized to false in every process, as global termination is detected when it turns to true in some process.

In accordance with the explanations of Section 1, PASSIVE can be set mechanically whenever P_j waits for a message or executes a final stop instruction and it can be reset after the arrival of a basic message, nevertheless 'manual tuning' is possible: It can be set as soon as P_j 'knows' that it will not send any more messages and its resetting might be left undone when the state of P_j indicates that further messages will eventually be received because the current state cannot be part of an (acceptable) final global state.

From the invariant and the time diagram it is easy to see that when the underlying distributed computation eventually terminates, the circulating vector will become the null vector shortly thereafter and termination will be detected by some process within one final round. In order to *prove* that the system is actually terminated at the instant when the circulating vector becomes the null vector, we examine the last cycle of the control message in the time diagram depicted in Figure 3 (recall that at least one round is completed).

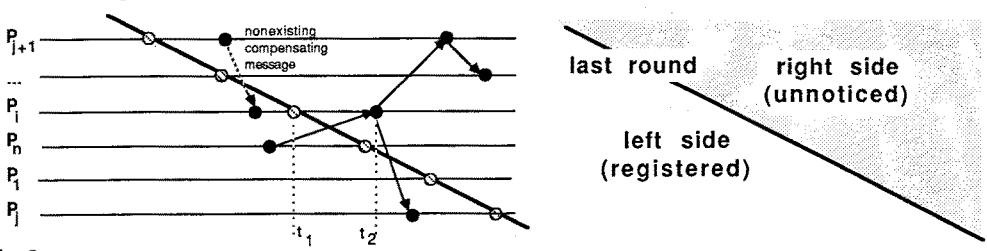

Fig. 3

Assume that the circulating vector becomes the null vector at some process P_j (i.e. the number of registered messages sent by each process equals the number of registered messages received) but that

there is activity at the right of the diagonal line representing the last control cycle. The first process (P_i in Figure 3) that becomes active after the last cycle at time t_2 can become so only due to an activating message which crosses the line (the control vector is only propagated when the process is passive). This means that the sending of the message is registered, but not its receipt. For C[i] to become 0 again, there must be a compensating message that crosses the line from the right side to the left side and whose arrival at P_i is detected, but whose sending goes unnoticed. Such a message must be received before t_1 (the time instant at which the control vector passes P_i) and consequently (because messages do not travel backwards in time) must also be sent out before t_1. However, every message sent before t_1 is registered by the circulating vector and no process activity exists at the right of the diagonal line before t_1, because the first activation takes place at $t_2 > t_1$. We conclude that a message crossing the diagonal line affects the vector counters in such a way that it is noticed. Hence there is no possibility of detecting 'false termination'.

We will come back to the vector algorithm in Section 7 where we make use of the fact that most of its properties are independent of the tour the control vector follows through the graph.

3. A toy problem - cryptarithmetic puzzles

In order to evaluate the different termination detection principles and other distributed control algorithms empirically, several experiments have been conducted with the INCAS research multicomputer system [NHM87]. An illustrative example is a parallel program which solves cryptarithmetic word puzzles by means of a *distributed constraint propagation scheme* [BEM86] based on an idea by Kornfeld [KOR81].

The problem consists of mapping the letters of three given strings onto the ten digits such that a correct addition results, e.g.:

LONGER	207563		EUROPE	290782
+LARGER	+283563	or	+EUREKA	+290234
---	---		---	---
MIDDLE	491126		SPIRIT	581016

To enable a parallel solution, each column is represented by a distinct process as depicted in Figure 4. Initially, all letters in all column processes are assigned the maximal set of digits {0,...,9}, and the carry-in and carry-out variables are initialized to {0,1}. A column process can receive messages from any other column (or from an external 'hypothesis generator') informing it of (new) constraints on letters or the carry values. Whenever a column receives a message containing new information it locally computes possible new constraints. One initial local transformation is performed when the process is established. New constraints on carry-in and carry-out values are sent to the right and left neighbor column respectively and new constraints on letters are sent to all columns which are interested in the information; a simple solution is to broadcast any new information and to let the receivers decide upon its usefulness. Notice that the FIFO property is not required and that the distributed computation behaves *nondeterministically* - therefore the number of messages is not exactly determined.

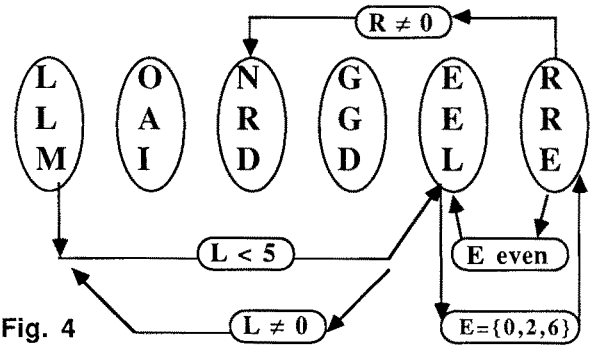

Fig. 4

Many variations of the general principle are possible, but the main point is that many constraint messages can be concurrently in transit and that the local computations of several column processes can be performed in parallel. However this does not mean, that the parallel solution is more efficient than the traditional sequential depth-first search with backtracking, generally a huge number of redundant computations are performed. The constraint propagation method can narrow the size of the search space sometimes drastically, but it may not be able to find a unique solution or a contradiction. If there is more than one possible solution it will never find any of them and the resulting sets of digits assigned to the letters will be supersets of all possible solutions. The resulting sets of the example of Figure 4 are L={1,2,3,4}, M={2,3,4,5,6,7,8,9}, R={1,3,5,6,8}, and E={0,2,6}; all other sets remain unchanged.

Since in general the constraint propagation *stagnates* without finding a complete solution, a *backtracking scheme* must be superimposed. The system then works in a sequence of two different *phases*: the parallel constraint propagation phase is used to prune the search space and when all constraint activity has quieted a backtracking hypothesis is generated. A hypothesis is like an ordinary constraint message; it is generated by an additional backtrack management process, which when detecting the termination of the constraint phase, sends it to all columns. In order to detect the end of a constraint propagation phase a *termination detection algorithm* is used. The principle of *multiphase algorithms*, where a new phase is started after the termination detection of the previous phase, has already been discussed in [CHM81].

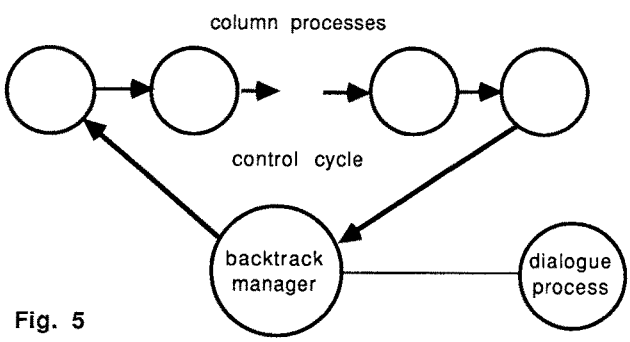

Fig. 5

Figure 5 shows the control topology of the puzzle-solving system, the control configuration is independent of the application topology. The *dialogue process* constitutes the interface to the user

where new puzzles are specified, the logical network is established, and the results are displayed. Here hypothesis can also be generated manually.

4. The distributed programming language CSSA

The puzzle-solving system including several different termination detection algorithms, among others the previously described vector method, has been realized in CSSA [MAB85]. CSSA (Computing System for Societies of Agents) is an experimental high-level programming language for expressing message-driven distributed application algorithms which involve many loosely linked cooperating tasks. Its underlying model of distributed computation is based on the notion of *actors* originally developed by Hewitt [HEW77].

CSSA provides a powerful set of language features for expressing communication and parallelism. Its sequential structures and data types are similar to those of Pascal, but concepts of modularization and data-abstraction have been combined in a homogeneous way to allow a structured implementation of distributed applications. Data values of each type, including those of recursively defined complex types (i.e. arrays, records, sets) and structures built up by dynamic records and pointers can be transmitted in messages.

Computations are performed by dynamically creatable *agents*, which are active objects that communicate with other agents solely by asynchronous message passing. An agent is an autonomous entity consisting of several clusters of *operations*. An operation can be activated by sending a message to the agent. Each agent processes only one message at a time without interruption and messages arriving at an agent while it is currently executing an operation are collected in a private *mailbox*. Execution of an operation may result in any number of messages being concurrently transmitted to other agents and many agents may be sending or receiving messages at the same time.

A cluster of operations together with local variable declarations is called a *facet*. Dynamic replacement of the current facet by another facet may change the behavior of an agent. Facets can also be set up recursively, the return to a previous facet then restores the old state allowing very simple realizations of backtrack algorithms. A facet has the following structure:

```
<var-decl>  /*global variables of the facet */
operation <name> <pattern> <assertion>
   .... /* local sequential code possibly with send and create instructions */
endoperation
operation  .... /* the next operation */
```

An agent is *passive* whenever it is not executing an operation while it implicitly scans the mailbox for an executable message. A necessary condition for the triggering of an operation is name equivalence. An operation describes the message it wants to receive by a pattern and an assertion. Among other things the *pattern* is used to break up composite data structures to extract pieces of the message and bind them to local variables; while the *assertion* allows the use of an arbitrary predicate on the values of the message and the variables of the agent. If the pattern-match succeeds and the assertion evaluates to true, the operation is executed with the actual variable-bindings, similar to the execution of a procedure. Otherwise, the message remains in the mailbox without any side effects and its match is retried at a later time.

By means of a specific agent connected to a terminal, the so-called *interface agent*, the user takes part in the distributed computation. It consists of a CSSA interpreter and is 'programmed' dynamically by the user during the computation. Similar to all other agents, the interface agent can send and receive messages and create new agents.

CSSA was designed and implemented as part of the INCAS project [NHM87]. A compiler running under UNIX generating code for a virtual stack machine was realized. A virtual machine forms the run time environment for an agent and provides extensive test and debugging facilities. A collection of virtual machines distributed over several hosts constitutes the distributed CSSA support and operating system. A complete CSSA system is now running on the INCAS research multicomputer consisting of several MC68000 based machines with a dedicated distributed operating system and on a network of UNIX machines connected by Ethernet using a TCP/IP based protocol. Several small applications and control algorithms are currently being implemented.

5. A distributed implementation of the puzzle-solving system

In the CSSA realization of the puzzle-solving system, the backtrack manager sends all its messages along the control cycle shown in Figure 5. The ring structure has the advantage in so far as the manager gets an implicit acknowledgement when all column processes have received the information, which is useful for *synchronizing* the columns. Backtracking is realized by recursive facets; the backtrack manager requests the columns to change their facet by two messages 'go_down' or 'go_up'. A 'report' message collects the sets of digits assigned to the letters, allowing the backtrack manager to get a *snapshot* of the global state. The snapshot is *consistent* when the constraint propagation phase has quieted. The backtrack manager can also propagate a hypothesis - an artificial constraint - and start the termination detection algorithm.

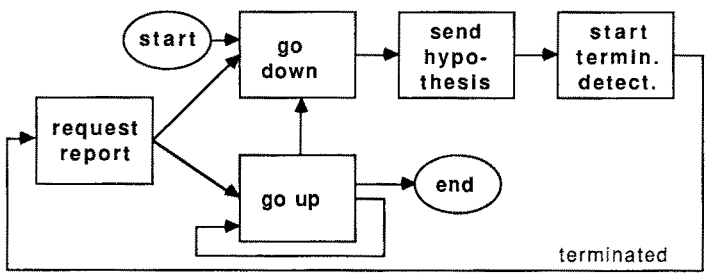

Fig. 6

The control flow of the backtrack manager is summarized in Figure 6. When it gets a snapshot after the end of a constraint propagation phase, either a solution is found, which is displayed, or a contradiction has been detected if some sets are empty, or some letters are still ambiguous. The backtrack manager reacts according to the backtracking principle and some hypothesis generating heuristics.

Each column process is realized by a CSSA agent and contains several operations for the acceptance of constraints and command messages in addition to an operation for detecting termination:

```
operation TERM_TEST (CTRL_VEC)
  assert LOC_VEC[MY_NO] + CTRL_VEC[MY_NO] ≤ 0
  LOC_VEC := LOC_VEC + CTRL_VEC ;
  if LOC_VEC = [0,...,0] then
    send TERM_TEST(LOC_VEC) to B_MANAGER ;
  else
    send TERM_TEST(LOC_VEC) to NEXT ;
    LOC_VEC := [0,...,0] ;
  endif ;
endoperation
```

The code of the operation should be self-explanatory. The main stratagem lies in the assertion: the TERM_TEST message is only accepted and the operation is only executed when the assertion evaluates to true, i.e. the acceptance of the control message is delayed until the agent has received a sufficient number of basic messages. This is equivalent to the guard V[j]≤0 used in Section 2. It should be annotated, that because of its message driven computational model, an agent is always passive at the moment of accepting a TERM_TEST message.

As soon as termination is detected by some column agent the backtrack manager is informed. No extra provision has to be made to ensure a complete first round, because before the termination test of a phase is started a hypothesis is sent to all columns. To initiate the first constraint propagation phase the backtrack manager sends a dummy hypothesis to the columns.

We do not advocate using a distributed system to solve puzzles as it was mainly used as a test case for our multicomputer system and several distributed control algorithms. The general problem is hard: While there are 'only' 10! different assignments of letters and digits to try, D. Eppstein [EPP87] has shown that a slight generalization, where the base of representation for the numbers is given as part of the problem (rather than always being decimal), is NP-complete. Nevertheless, one may be curious whether the distributed constraint propagation solution is more efficient than a pure sequential backtracking method. The efficiency of the distributed solution naturally depends on various system and implementation characteristics. Our experiments have shown that when using real-time as an efficiency measure in most cases the sequential method is faster, but for some puzzles the distributed solution is more efficient. For puzzles with 5 to 10 columns typically 20 to 80 basic messages are generated in one phase and a column agent usually performs less than 10 transformations per phase. Compared to the traditional sequential backtracking algorithm where about 1000 to 5000 hypotheses are generated, the constraint-based solution generates only about 20 to 100 hypotheses when using a very simple hypothesis generating heuristic [BEM86]. This shows that the search tree is pruned drastically and that the distributed constraint propagation method can be a valuable principle for large search spaces in more serious applications. Of course, the constraint propagation principle can also be used in a sequential algorithm where it might be possible to avoid some redundant computations, but a potential performance gain is mainly to be expected from the parallel computation of the local transformations.

Several variants of the general principle have been implemented, among them a distributed backtracking version where several hypotheses are tested in parallel, and variants with other termination detection algorithms, e.g. with problem oriented termination conditions.

6. An empirical evaluation of the vector algorithm

How efficient is the vector termination detection algorithm? We know from Section 2 that the *worst case message complexity* is O(mn) for the ring configuration, and it is also the *worst case time complexity*. Other configurations with lower message complexity and parallel traversal schemes yielding lower time complexity will be considered in the next Section. However, we expect the worst case situation in which only one basic message is received in every round to be very unlikely and we would like to know the 'typical' complexity of the algorithm. The behavior of the termination detection algorithm is highly dependent on the communication pattern of the underlying computation and the communication delay of the basic messages. The execution speed of local computations is also relevant but in the message driven model this can be subsumed to the communication delay, at least theoretically. Unfortunately there does not seem to be any canonical probability distribution on the delays and the message generation patterns (i.e. the number of messages sent as a reaction to a message received) which would enable us to compute the average complexity. We are therefore limited to statistical results of typical distributed computations.

To evaluate the vector termination detection method the puzzle-solving system was used. A large number of puzzles have been executed and for every phase the basic messages and the control messages have been counted. The following table shows typical results of the ring based termination detection algorithm for some selected puzzles:

puzzle	initial hypotheses	basic msgs m	control msgs	theor. max (n+1)(m+1)
ABC+ABC=DDEC	-	30	**7**	155
LONGER+LARGER=MIDDLE	-	47	**9**	336
RIVER+WATER=SHIPS	-	38	**11**	234
	V=4, E=3	121	**12**	732
DONALD+GERALD=ROBERT	-	47	**8**	336
	A=4, B=3	153	**13**	1078
ABCDEAABCDEA+	-	135	**15**	1768
FGHDEAFGHDEA=	A=5 (phase 2)	319	**22**	4160
HBIGHJHBIGHJ	D=4, I=3, H=7	468	**20**	6097
ABCDAABCDA+EFGDAEFGDA= HIBJHHIBJH	C=4, D=3	429	**23**	4730

Various runs of the same puzzle usually produce slightly different figures since the computation is nondeterministic. One has to be careful when assessing and generalizing the results; although a puzzle computation usually shows a rather 'random' behavior with communication peaks alternating with periods of low communication activity we have no actual justification to regard it as a 'typical' distributed computation. However, in all runs the number of control messages was far less than could be expected from the worst case behavior (n+1)(m+1); in fact $\sqrt{m}$ seems to be a rather good approximation. Basic messages and control messages are treated with equal priority. Even less control messages would be needed if we could give control messages a *lower priority*; in which case, whenever

several messages are in the mailbox of an agent, a termination test message will only be selected if no basic message is acceptable. The unbounded version of the algorithm without the assertion to delay 'early' control vectors did not perform significantly worse. This can be explained by the fact that in our system messages are mostly delivered in the same order they have been sent.

To gain more confidence in the low empirical message complexity we are currently implementing the vector algorithm in other small applications. Similar results have already been obtained from a distributed program for computing the greatest common divisor of several numbers. A more ambitious project consists in the realization of a general testbed for distributed control algorithms where various network structures and stochastic communication patterns can be specified. First experimental results with that system confirm the low message complexity of the algorithm.

Recently a similar termination detection algorithm has been implemented in the distributed run time system of CSSA. Various pragmatic conditions had to be taken into consideration, e.g., input from the environment, restart of the algorithm after termination detection, dynamic creation of agents. Also, several optimizations could be applied; among other things internal message buffers are regarded as 'processes' which are only passive when they are empty. Whenever the run time system detects termination, a synchronous notification message is automatically sent to all agents which allows any agent to start the next phase.

7. The vector algorithm revisited - important variants

In the puzzle-solving system any column that detects termination of a constraint phase directly informs the backtrack manager. Other columns normally do not notice the fact until they 'suddenly' get a request message from the backtrack manager. A review of the algorithm as it was described in Section 2 shows that only one process eventually knows that the system has terminated. However, by simply removing the 'else' in line 3 of the termination demon the null vector will continue to circle and inform all processes. To stop the vector after one round the guard should be extended by another condition 'and not TERMINATED'. One may be tempted to 'approximate common knowledge' by letting the vector accomplish just one more round to inform each process that every process knows that the system has terminated...

Some additional measures have to be taken providing *every* process should have the possibility to restart the system in a multiphase application and not just one single process as in the puzzle-solving system. After restarting the system (only if TERMINATED is true!) a process sets its local predicate variable PASSIVE to false and sends out a *start control message* which travels around the control ring resetting all TERMINATED flags to false. This enables the control vector to leave the process where it was hidden (the process with HAVE_VECTOR = true). Some care has to be taken because several processes may start the system independently and the revitalized control vector can overtake start messages. A preventive countermeasure to possible confusion consist in considering a process active until it gets back its own start message. Since the control vector must perform at least one round (which has to be ensured by some simple additional mechanisms not discussed here) before setting any TERMINATED flags to true, any control vector which is too fast will be blocked in its first round at the starting process until all processes have been informed about the new phase. Another potential problem could exist with processes which restart the system too early, before all other processes are informed about the termination of the previous phase. This problem can be overcome by numbering

the phases and rejecting control messages of a later phase. The problem of independent restart is reminiscent of the *distributed election* problem.

When considering *control topologies other than rings* it is important to realize that the proof of the main property of the algorithm (that the distributed computation is actually terminated when the circulating control vector becomes the null vector) is not limited to rings (or Hamiltonian cycles) where each process is visited exactly once. 'Cycle' can be substituted by any serial *traversal scheme* visiting all nodes at least once and in any such traversal (with the possible exception of the first one) it is guaranteed that at least one basic message is received. (To keep the invariant correct it is important however that the local vectors are reset to [0,...,0] when their values are accumulated by the control vector). Since it is reasonable to assume that the network is strongly connected (otherwise there exists a node which can never know whether all other nodes are passive), there exists always a closed traversal scheme - additional communication channels solely for control messages are therefore not necessary. However, the efficient construction of short closed traversals in directed networks is a non-trivial problem [GAA84]. Kutten [KUT87] also investigates this problem; his serial "non-retreating traversal algorithm" has an expected message complexity of $O(n\sqrt{ne})$ for (random) graphs of n nodes and e edges. The PIF algorithm (Propagation of Information with Feedback) presented in [TEL87] can also be used to propagate a control vector through a directed network, however its message complexity is $O(n^3)$ and an upper bound for the diameter of the network must be known to all processes.

The traversal sequence of the control vector C needs not to be the same in every traversal. However, to guarantee the progress of the algorithm every process P_j for which C[j]>0 must eventually be visited. This suggests an interesting traversal strategy for *complete networks*: The control vector starts with C=[1,...,1] and the local vector variables V of any P_j have all components initialized to 0 except V[j], which is set to -1. Until it becomes the null vector, C is propagated to *some* process P_i for which the i-th component of the accumulated vector is greater than 0. This can be interpreted as if the control vector *chases* previously sent basic messages. Definitely n+m control messages at the maximum are used. For normal computations, where every process receives at least one basic message or where m>>n, this is *(almost) optimal*; Chandy and Misra have shown that any distributed termination detection algorithm uses at least m control messages in the worst case [CHM86].

The solution for complete networks also suggests a variant for a *star network* with bidirectional channels using at most 2(n+m) control messages: instead of propagating the control vector directly to some other process, it is always returned to the central node. The central node sends the 'deficit' D=C[j] to some process P_j for which C[j]>0. According to the previously described strategy, P_j sends back its vector V when it is passive and D+V[j]≤0. P_j resets its local variable V and the central node accumulates the values by C←C+V.

Instead of sequentially visiting the nodes, the central node can request *in parallel* all processes P_j for which C[j]>0 to send back their vector V. After having received the answers to all requests it starts a new 'round' until C=[0,...,0]. This solution seems particularly attractive if the request together with the deficits D[j], where D[j] is the last rounds' C[j], can be *broadcast* to all processes.

Both variants, the sequential and parallel version of the star configuration have been implemented in the puzzle-solving system with the backtrack manager acting as the central process. The following table compares the results for the same puzzle instances as shown in the previous table. Notice that due to the nondeterministic behavior of the computation the number of messages for the ring topology are not always the same as those listed in the first table.

basic messages	control messages		
	ring	star (seq.)	star (par.)
32	7	10	11
47	11	12	18
38	11	10	14
136	13	14	20
47	8	11	11
138	13	14	17
135	15	25	29
386	19	30	34
501	19	30	35
436	17	26	38

We observe that usually the sequential and parallel star configurations use more control messages than the ring based variant (1 : 1.32 : 1.56 on the average for all our experiments), although their worst case message complexity is only 2(m+n) compared to (m+1)(n+1). Again, one has to be extremely careful when generalizing this result since the number of control messages and the relative performance of the algorithms is dependent on the application and various characteristics of the underlying system. Nevertheless, the experiments show that under some conditions the theoretical bounds for the number of messages are of little importance, therefore in such cases an unbounded but simpler algorithm could be more appropriate (e.g. [KUM85], [MAT87a]).

Another canonical control structure for a termination detection algorithm when considering networks with *bidirectional channels* is a *fixed spanning tree* ([FRA80], [FRR82], [TOP84]). A wave of deficit control vectors D (copies of the accumulated control vector of the previous round) sent out by the root process moves through the tree. Every node stores the deficit vector D and propagates D+V. Eventually the deficit wave reaches the leaves where the request messages are turned into 'echoes' moving upwards. The wave of deficit vectors moving down the tree is only started or accepted and propagated by any node P_j if $D[j]+V[j] \leq 0$. Echoes carry the accumulated control vectors of the subtrees; a root of a subtree combines all its subtree control vectors including its own vector V. It sends an echo vector back only when D plus the vector components of all the nodes in its subtree are 0 or negative, otherwise the respective subtrees are revisited with a new deficit vector. The previously described star configuration can be seen as a degenerated tree of depth one.

In trees, it is also possible to *split* the 'circulating' vector and visit the subtrees in parallel: When a vector is received from the parent node it is added to the local vector. Then for each subtree those components of the vector are *extracted* which represent nodes of the subtree. If at least one of those components is non-zero a new vector is constructed (with all other components set to 0) which is sent to the root of the subtree. When the echoes have been received, and its values have been accumulated, this procedure can be repeated until no more subtrees can be revisited. Then the accumulated vector is sent back. Technically, this scheme is equivalent to the previously described principle using deficit vectors on trees. Notice that by splitting the vector, the invariant is kept valid. The *splitting vector technique* can be regarded as a generalization of the vector principle, the previously described realizations for rings and star networks are merely variants of special cases.

Most interesting are traversal schemes for undirected networks based on *echo algorithms* which do not need any additional communication channels or predefined control structures. The classical parallel traversal scheme with a given initiator has time complexity proportional to the diameter of the network and uses $O(n^2)$ messages ([CHA82], [SEG83]), but improvements for dense networks and an O(n) sequential depth first traversal scheme for networks in which nodes know the identities of their neighbors have recently been presented ([HMP87], [HMR86]). Notice that the neighborhood knowledge

can easily be obtained by exchanging 2e messages once with e denoting the number of edges ($e<n^2$). The $O(n^2)$ messages of this initialization phase can be subsumed under the O(mn) messages of the termination detection algorithm for most reasonable cases (e.g. $m \geq n$). The echo traversal schemes dynamically construct a spanning tree, superimposed termination detection algorithms are therefore similar to the previously sketched fixed spanning tree variant with the exception that a dedicated revisiting of subtrees is a priori not possible. Echo based termination detection algorithms, which use a scalar counter instead of a vector, are discussed in [MAT87b].

For strongly connected *directed* networks the use of trees is slightly more complicated. For a given root a directed out-tree and a different in-tree can be constructed without adding new communication channels [GAA84], [KUT87]. The root uses the out-tree to propagate the deficit vector to every node. At the leaves of the in-tree it evokes a wave directed towards the root, while inner nodes of the in-tree accumulate the results of the subtrees. The nodes delay the propagation of the wave on the in-tree until at least as many basic messages as indicated by the deficit vector have been received.

Instead of accumulating the local vectors it might be possible to collect the values *without resetting* them to [0,...,0]. This is feasible if the values of every process are collected exactly once during a traversal. Appropriate configurations are rings, but also trees (either fixed or generated implicitly by echo algorithms). On trees and echo traversals the value of a node is collected just before the control vector is sent back to the father. Since no local variables in the processes are changed, several independent control waves can be concurrently active in such a *reentrant* variant. For a potential application consider a ring where any process suspecting termination could send out a collecting vector (initialized to [0,...,0]) independently of all other processes. When the process gets back its own vector it can announce global termination if it is the null vector. Notice that this is a completely *symmetric* realization because there is no privileged process. However, if no information from the previous round is carried over to the next round the number of control rounds is not bounded. This can easily be seen by considering the ring configuration and a very slow basic message that travels in the opposite direction than the control messages. For rings a simple solution consists in alternating the direction of the control cycle, then in every other round at least one basic message is received.

For bounded symmetric versions on other topologies it is necessary to carry over some information from the previous round by splitting the control vector C[1:n] into two components S[1:n] and R[1:n] (C=S+R) and adding a third vector L[1:n]. S[k] denotes the number of basic messages sent, R[k] the number of basic messages received. L is a copy of the vector S of the last round (or [0,...,0] in the first round). Instead of C, a 3-tupel (L,S,R) is propagated (S and R are initialized to [0,...,0] when starting a round). A process P_i accepts the control vector (L,S,R) only if $V[i]+L[i] \leq 0$. Then S and R are updated according to $S[k] \leftarrow S[k]+V[k]$ for all $k \neq i$ and $R[i] \leftarrow R[i]+V[i]$. After the initiator has gotten back its vector (L,S,R) and added its own values to S and R it announces termination if S+R=[0,...,0]. By virtue of the guard $V[i]+L[i] \leq 0$ it is guaranteed that at least one basic message is received in every round (except for the first round). L[k] represents the knowledge of the number of messages sent to P_k which has been gathered during the previous round; at least as many messages must have been received in the next round.

The use of fixed length vectors is inappropriate for *dynamic systems*; however, sets of pairs consisting of process identification and counter value could be used instead. To cope with the dynamics, a process creating a new one could send a virtual message to this new process, to itself, or to some other dedicated process by setting the appropriate component in its local vector. The virtual message is

only considered as being received when the new process has been integrated into the control configuration. More control is needed for terminating processes.

8. Conclusion

We have proposed a new algorithm for detecting termination of distributed computations. Compared to most other solutions the vector algorithm is not restricted to synchronous communication or FIFO channels. The message complexity is bounded by O(mn) for rings, trees and sequential depth first traversal schemes and by O(m) for stars and complete graphs; but our experimental results have shown that usually far less messages are generated than can be expected from the worst case. A drawback of the method is the length of the control message, it consists of a vector of length n. Only three other methods based on the same general model of distributed computation are known:

a) The *sceptic algorithm* using scalar counters ([KUM85], [BMR85], [MAT87a]): The message complexity is not bounded and at least two full rounds are required.

b) The *time algorithm* ([LAI86], [BMR85], [MAT87a]): It has message complexity O(mn) and each basic message must be augmented with a time stamp of a virtual clock.

c) The *diffusing computations scheme* ([DIS80], [SHF86]): By a signaling scheme each basic message is acknowledged, resulting in exactly m control messages. While the worst case is optimal, there is no better average case.

The general principle of our method, which consists in counting the basic messages individually per process, gives rise to a whole *class* of termination detection algorithms with bounded message complexity. The scheme can be superimposed on different traversal algorithms, resulting in sequential and parallel variants for various topologies. It is simple, readily implementable, and the local computations are short and efficient. Symmetric and reentrant versions allow any process to start a termination test independently of all other processes.

The results of our experiments indicate that an empirical evaluation of distributed control algorithms is useful, and that the worst case message complexity should not be the only evaluation criterion when assessing distributed algorithms. The number of control messages is often highly dependent on the communication patterns of the underlying computation and the characteristics of the communication system. We will continue our experiments to see how sensitive different termination detection algorithms and other control algorithms are with respect to various characteristics of the environment.

References

[BEM86] BEILKEN C., MATTERN F. (1986) *Verteiltes Problemlösen am Beispiel von Zahlenrätseln – Ein Experiment mit CSSA.* Technical Report SFB124-29/86, Department of Computer Science, University of Kaiserslautern, West-Germany

[BMR85] BEILKEN C., MATTERN F., REINFRANK M. (1985) *Verteilte Terminierung – ein wesentlicher Aspekt der Kontrolle in verteilten Systemen.* Technical Report SFB124–41/85, Department of Computer Science, University of Kaiserslautern, West-Germany

[CHA82] CHANG E.J.H. (1982) *Echo Algorithms: Depth Parallel Operations on General Graphs.* IEEE Transactions on Software Engineering SE-8:4, pp. 391-401

[CHM81] CHANDY K.M., MISRA J. (1981) *Asynchronous Distributed Simulation via a Sequence of Parallel Computations.* Comm. of the ACM 24:4, pp. 198-205

[CHM86] CHANDY K.M., MISRA J. (1986) *How Processes Learn.* Distributed Computing 1, pp. 40-52

[DIS80] DIJKSTRA E.W., SCHOLTEN C.S. (1980) *Termination Detection for Diffusing Computations.* Information Processing Letters 11:1, pp. 1-4

[EPP87] EPPSTEIN D. (1987) *On the NP-Completeness of Cryptarithms.* ACM SIGACT News 18:3, pp. 38-40

[FRA80] FRANCEZ N. (1980) *Distributed Termination.* ACM Trans. on Prog. Lang. and Sys. 2:1, pp. 42-55

[FRR82] FRANCEZ N., RODEH M. (1982) *Achieving Distributed Termination without Freezing.* IEEE Transactions on Software Engineering SE-8:3, pp. 287-292

[GAA84] GAFNI E., AFEK Y. (1984) *Election and Traversal in Unidirectional Networks.* Proc. of the 3rd Annual ACM Symposium on PODC, pp. 190-198

[HEW77] HEWITT C. (1977) *Viewing Control Structures as Patterns of Passing Messages.* Art. Intell. 8, pp. 323-364

[HMP87] HELARY J.-M., MADDI A., PLOUZEAU N., RAYNAL M. (1987) *Parcours et apprentissage dans un reseau de processus communicants.* Technique et Science Informatiques 6:2, pp. 127-139

[HMR86] HELARY J.-M., MADDI A., RAYNAL M. (1986) *Calcul distribue d'un extremum et du routage associe dans un reseau quelconque.* Technical Report 516, INRIA, France

[KOR81] KORNFELD W.A. (1981) *The Use of Parallelism to Implement a Heuristic Search.* Proc. of the International Joint Conference on Artificial Intelligence, pp. 575-580

[KUM85] KUMAR D. (1985) *A Class of Termination Detection Algorithms for Distributed Computations.* In: Maheshwari N. (ed) Fifth Conference on Foundations of Software Technology and Theoretical Computer Science, Springer-Verlag, LNCS 206, pp. 73-100

[KUT87] KUTTEN S. (1987) *Stepwise Construction of an Efficient Distributed Traversing Algorithm for General Strongly Connected Directed Graphs.* Technical Report 431 (draft), Technion - Israel Institute of Technology, Computer Science Department, Haifa, Israel

[LAI86] LAI T.-H. (1986) *Termination Detection for Dynamic Distributed Systems with Non-first-in-first-out Communication.* Journal of Parallel and Distributed Computing 3, pp. 577-599

[MAB85] MATTERN F., BEILKEN C. (1985) *The Distributed Programming Language CSSA - a Very Short Introduction.* Technical Report 123/85, Department of Computer Science, University of Kaiserslautern, West-Germany

[MAT87a] MATTERN F. (1987) *Algorithms for Distributed Termination Detection.* Distributed Computing 2:4 (to appear)

[MAT87b] MATTERN F. (1987) *Asynchronous Distributed Te rmination – Parallel and Symmetric Solutions with Echo Algorithms.* Technical Repor . SFB124-21/87, Department of Computer Science, University of Kaiserslautern, West-Ger many

[NHM87] NEHMER J., HABAN D., MATTERN F., ROMBACH). , WYBRANIETZ D. (1987) *Key Concepts of the INCAS Multicomputer Project.* IEEE Tr nsactions on Software Engineering SE-13:8, pp. 913-923

[SEG83] SEGAL A. (1983) *Distributed Network Protocols*. IEEE Transactions on Information Theory IT-29:1, pp. 23-35

[SHF86] SHAVIT N., FRANCEZ N. (1986) *A New Approach to Detection of Locally Indicative Stability*. Technical Report RC 11925, IBM Th. J. Watson Research Center, Yorktown Heights, USA

[TAL86] TAN R.B., VAN LEEUWEN J. (1986) *General Symmetric Distributed Termination Detection*. Technical Report RUU–CS–86–2, Computer Science Department, University of Utrecht, Utrecht, The Netherlands

[TEL87] TEL G. (1987) *Directed Network Protocols*. In: Gafni E., Raynal M., Santoro N., van Leeuwen J., Zaks S. (eds) Proc. 2nd Int. Workshop on Distributed Algorithms, Springer-Verlag, LNCS

[TOP84] TOPOR R.W. (1984) *Termination Detection for Distributed Computations*. Information Processing Letters 18:1, pp. 33-36