Proc. Fourth ACM Symposium on Principles of Distributing Computing, Minacki, Canada, pp. 287-292, 1985

Terminating Iterative Solution of Simultaneous Equations* in Distributed Message Passing Systems

by

B. Szymanski, Y. Shi and N. Prywes

Department of Computer and Information Science
University of Pennsylvania, Philadelphia, PA 19104, USA

Abstract

An efficient algorithm for synchronized termination of iterative solution of simultaneous equations in a distributed message passing system is presented. The algorithm is based on an assumption that distributed processors are connected via unidirectional channels into a strongly connected network, in which no central controller exists. The number of processors and the network configuration are not known a priori.

1. Statement of the Problem and Comparison with Other Solutions

Simultaneous equations are widely used for modelling of large scale systems in science and engineering. In many applications, a modelled system is composed of local models highly interdependent on each other. In particular, a cooperative computation consists of an aggregate of local modules being developed and maintained by a number of organizationally and geographically dispersed groups [5]. Integrating local models into a global system on a single computer would require solving iteratively simultaneous equations from all the local models. It would result in a large sparse coefficient matrix and a system of equations which is inefficient to solve. Instead, a distributed solution can be used, where the dense clusters of variables in each local model are solved iteratively locally and the local solutions are communicated to other models repeatedly until global convergence of the communicated variables is attained. Distributed solution supports the geographical dispersal of specialists, the testing and integration with real life data, and finally the parallelism needed to complete computation in a reasonable time.

Distributed solutions should allow for independent development of local models, their testing without linkage to other models and then experimentation with progressively growing numbers of models being connected together. To achieve that, the use of a very high level language MODEL [6] along with underlying compiler and configurator has been proposed [7]. The compiler generates programs for individual models from specifications independently developed by each local group of users. Based on simple, graph-oriented description of a network of models, the configurator

*Work performed with support from the Office of Naval Research under contract #N00014-76-C-0416

generates software necessary to link together local models.

Each node of the network represents a set of simultaneous equations being solved iteratively. Local solution of a node is communicated as an initial value of the next iteration to the successor nodes, which in turn, communicate their solutions as initial values to their successors and so on. As the entire computation represents a set of simultaneous equations the network is a strongly connected directed graph. Each node monitors relative difference between values communicated to it in subsequent iterations. If relative difference is within predefined limits, the node assumes that local convergence has been achieved. However, it further performs solving its local equations until all nodes converge (the global convergence is attained) and the entire computation can be terminated.

There is a need for an algorithm, which attached to each node could be able to detect that global convergence has been achieved. Obviously, such an algorithm has to pass tokens reporting states of nodes in the network. As computers on which local models are implemented can be geographically dispersed we are interested in an algorithm which uses only existing unidirectional communication traffic of local solutions. To support experimentation with different model linkages, we assume that network configuration may not be known a priori. We require also that the algorithm stops computation in such a way that no tokens are passed to nodes already deactivated.

The formulated problem belongs to a class of so-called distributed termination problems [1,2,8], which in the most general form can be described as follows. There are given n processors connected into a network and performing a common computation. The system is asynchronous (processes run at arbitrary finite nonzero speeds, message communications take arbitrary finite time, messages are not lost or corrupted and they are delivered in order). The computation has to be stopped when all the processes meet their local termination conditions. Each process is able to determine only its own termination condition. The task is to design an algorithm which detects an instant in which the entire computation can be stopped.

The problem has many variations depending on the type of network involved (directed or undirected [1,2] graph), state of processes which satisfy local termination conditions (active or idling [1,3]) and restrictions imposed on the algorithm. As explained above, in this paper we assume that processes exchange messages via unidirectional channels which connect them into a network represented by a strongly connected directed graph. Each process outputs its (i+1)th output along every outcoming communication channel only after receiving ith input along every incoming channel. This property introduces some degree of synchroneity in the main computation (all processes are engaged in the computation of the same iteration step).

A desirable solution should also satisfy following restrictions:

- The modification of each process is independent of the number of processes or network configuration. It means that exactly the same termination detection algorithm is attached to each process.
- No new communication channels between processes are added. Since communication channels are assumed to be unidirectional, it means that we can not propagate termination signals forward and backward, like in [3].
- When any of the processes terminates no other process can pass a message to it (the termination is synchronized).

No solution of distributed termination problem published to date [1,2,3,8] satisfies all the specified requirements.

They require bidirectional communication channels and a priori knowledge of network configuration. On the other hand they allow totally asynchronous main computation.

2. Termination Detection Algorithm

In the sequel we will use standard terminology of graph theory [4]. Let x,y,z,v ... stand for nodes of the network and Z^+ for the set of nonnegative integers. The length of the shortest (directed) path from a node x to y is called a distance from x to y and is denoted by L(x,y). The greatest distance between any two nodes in the network, called diameter, is denoted by D.

The termination detection algorithm involves passing tokens through the network and evaluating node states. Tokens are labelled by integer values ranging from 0 to D+1. They are transmitted as a part of communication traffic generated by the main computation. Node states are determined based on local convergence and labels of received tokens. The value of the node state is used as a label of output tokens.

Let C_x^i denote a predicate indicating whether the local convergence is attained in node x for iteration i. We assume that initially all processes are not convergent, i.e. C_x^0 is false for any node x.

By I_x^i we will denote the minimum label of tokens received at node x in the ith input.

Definition 1. For any node x andy any iteration step i the node state S_x^i is defined as:

$$S_x^i = \begin{cases} 0 & \text{- if } C_x^i \text{ is false} \\ I_x^i+1 & \text{- if } C_x^i \text{ is true and } I_x^i < S_x^{i-1} \\ S_x^{i-1}+1 & \text{- otherwise} \end{cases}$$

Tokens output from the node x in the ith iteration are labelled by S_x^i. (These tokens are received by successors of node x as a part of their (i+1)th input). They are sent out after the node received all its ith input, but before reading the next input.

Informally speaking a node x generates tokens labelled by 0 only when C_x^i is false (i.e. its computation did not converge yet). When all nodes' computations converged, no tokens labelled by 0 are generated. Therefore labels of transmitted tokens grow in value, as do the node states. We show below that the network diameter is a threshold value which a node state can exceed only after all nodes' computations have converged. All nodes reach that state with the same iteration number.

Theorem 1. For any node x and the iteration step number i such that $S_x^i > 0$ if for some node y, $L(y,x) < S_x^i$ then

$$(\forall j \in Z^+ : j < S_x^{i-L(y,x)}) \; S_x^{i-j-L(y,x)} > 0$$

Proof. The theorem holds for step i=0, since, by definition, $S_x^0=0$, for any node x. Suppose that the theorem does not hold for a node y. Let i0>0 denote the smallest step for which this happens, and z denotes a node violating the theorem. Thus, we have

$$(\exists j \in Z^+ : j < S_y^{i0-L(z,y)}) \; S_z^{i0-j-L(z,y)} = 0.$$

Let j0 denote such j. Since the theorem holds for i0-1, we have

$$(\forall j \in Z^+ : j < S_y^{i0-1-L(z,y)}) \; S_z^{i0-1-j-L(z,y)} > 0.$$

Since $S_y^i \leq S_y^{i-1}+1$, for any index i>0 and any node y, then j0 may be only equal to $S_y^{i0-L(z,y)}$. Now, traversing the shortest path from z to y and starting at the step i0-L(z,y) and with the state $S_z^{i0-L(z,y)}=0$, we reach y at the step i0 with the input such that $I_y^{i0} \leq L(z,y)$. But the definition of the state implies that $S_y^{i0} \leq I_x^{i0}$, which contradicts the assumption that

$S_x^{i0} > L(z,y)$. □

If for any node x and index i, $S_x^i > D$ then from the theorem 1 we conclude that the inequality $S_y^{i-D} > 0$ holds for every node y in the network (since $L(y,x) \leq D$) and, hence, the entire computation is in a stable state.

Theorem 2. For any node x and any index i there is such a node y that $S_y^{i-j} = 0$ for $j = S_x^i$.

Proof. If $S_x^i = 0$ then theorem hold by setting y=x. When $S_x^i > 0$ we can always construct a sequence of nodes $y_0 = x, y_1, .. y_k$, $k = S_x^i$ such that for p=1,2...k:

$$S_{y_p}^{i-p} = S_{y_{p-1}}^{i-p+1} - 1$$

To do that, let's consider evaluation of $S_{y_{p-1}}^{i-p+1}$. If $S_{y_{p-1}}^{i-p+1} = S_{y_{p-1}}^{i-p} + 1$ then we set $y_p = y_{p-1}$. Otherwise y_p is equal to the the node which sends the token with the minimum label in the (i-p+1)th input to the node y_{p-1}. The last node in the constructed sequence, i.e. $y_{S_x^i}$, satisfies the condition of the theorem. □

Theorems 1 and 2 show that a node state represents the distance (measured in the length of the shortest path and/or the number of iterations) from the given node x to the closest node with divergent computation.

From these two theorems we can also conclude that all nodes of the network reach the state D+1 with the same iteration number. Therefore reaching this state can serve as an indicator of the global convergence and be used for termination of the entire computation. It is easy to verify that any value greater than D can be used as such indicator, leading however to unnecessary continuation of the main computation for iterations D+2, D+3, etc. D+1 is in fact the smallest threshold value independent of the pattern of getting nodes ready to stop.

The implementation of the algorithm is straightforward. It involves only extending all the messages being passed by the main computation with a single field for a token and calculating nodes states according to the definition 1.

To terminate the main computation, the algorithm requires D+1 iteration steps from the instant at which the last process has become convergent. Let v denote the last process not yet convergent. Under assumed communication protocol the information about a change of state of node v reaches a node x after L(v,x) iteration steps. Therefore information about the change cannot reach all the nodes earlier then after max(L(v,x)), where maximum is taken over every node x of the network. This value is equal to D at least for some v. Thus, we conclude that the number of steps of the proposed algorithm is equal to the minimum number of steps required in the worst case by any algorithm. The number of tokens passed by the algorithm is equal to $(i_{max}+D+1) \times E$, where i_{max} stands for the number of iterations in which the last process becomes convergent and E denotes the number of edges in the network. However, each token is a part of a message passed by the main computation, therefore more important measure of communication overhead is the total increase in communication traffic incurred by the algorithm. Each token carries only one label represented by an integer from 0 to D+1. Thus the total communication increase measured in bits is equal to $(i_{max}+D+1) \times E \times \log_2(D+1)$.

3. Finding the Diameter of the Network

Lack of knowledge of full network configuration at compilation time implies that the network diameter may be unknown until run-time. We may request to provide it to each node during run-time as an input data, or we may

use some upper limit of the actual diameter instead. The latter solution is particularly convenient when computations are performed on different subnetworks of one fixed network whose diameter is known. However, in a general case, we need an algorithm for finding the diameter of the actual network configuration from any node.

Let M denotes a connectivity matrix of the network with n nodes. The network diameter is the smallest exponent D such that the sum $M^1+M^2+...+M^D$ yields the matrix with all entries equal to 1. We will construct in each node a single vector representing a row of the kth power of M for the kth algorithm step. Let $m_{x,y}$, x=1,2...,n, y=1,2...,n, denote the elements of the connectivity matrix M. $m_{x,y}=1$ means as usual that the node x is a predecessor of the node y. The vector V_y^k representing the kth power of M in the yth node is equal to

$$V_y^k=\sum_{z=1}^{n} V_z^{k-1}\times m_{z,y}=\sum_{m_{z,y}=1} V_z^{k-1}$$

Therefore, constructing such a vector in the given node requires only receiving vectors of the previous power of M from the node's predecessors. The existing communication channels can be readily used to carry that task.

To further decrease the communication traffic we can pass only this part of a vector V_z^{k-1} which contains 1's not passed in the previous steps and not to store 0's at all. Thus, in each step k we pass names of nodes by which the vector V_z^{k-1} was extended in the previous step.

A node y can recognize that it has constructed the entire vector V_y^k when in some iteration k all the received node names have already reached the node y before. Such step number k represent also the length of the longest path ending at the node y. For any node x we will denote that value by D_x and call it a relative diameter of the node x.

In quite a similar way to the termination state token passing, the nodes can pass also tokens representing the biggest distances found at the given iteration step. Each node sends out tokens labelled by the maximum value among distances received at its input and its own relative diameter (or step number, if the relative diameter has not been evaluated yet). A relative diameter of a node y is used to find out the network diameter according to the following theorem.

Theorem 3. The network diameter D is equal to the first such label of the node's output token that repeats after the step $2\times D_y$.

Proof. Let v denote such a node that $D_v=D$. Since $L(v,y) \leq D_y$ then in the step $D_y+L(v,y)+1$ the token labelled by D_y+1 sent out from v reaches y. From that step on labels of tokens output from y will grow until they reach D. □

As we use the existing communication channels to accomodate traffic of tokens created by this algorithm we can attached these tokens to the messages communicated by the main computation. In such case $2\times D+1$ is the first iteration step at which the main computation can be stopped.

References

1. Dijkstra E.W., Feijen W.H.J. and van Gasteren A.J.M., "Derivation of a Termination Detection Algorithm for Distributed Computations," Information Processing Letters, Vol.16, No,5., 1983, pp. 217-219.

2. Francez N., "Distributed Termination," ACM Trans. on Programming Languages and Systems, Vol.2, No.1, 1980, pp. 42-55.

3. Francez N. and Rodeh M.,"Achieving Distributed Termination Without Freezing," IEEE Trans. on Software Engineering, Vol.SE-8, No.3, 1982, pp. 287-292.

4. Harary L.,Graph Theory, Addison-Wesley, Reading, Mass., 1972.

5. Prywes N. and Pnueli A., "Automatic Programming In Distributed Cooperative Computation," IEEE Trans. on Systems, Man and Cybernetics, April 1984.

6. Prywes N. and Pnueli A., "Compilation of Nonprocedural Specification Into Computer Programs," IEEE Trans. on Software Engineering, Vol.SE-9, No.3, 1983, pp. 267-279.

7. Prywes N., Szymanski B., and Shi Y., "Nonprocedural-Dataflow Specification of Concurrent Programs," Proceedings of COMPSAC 83, p. 287-294, Chicago, November 1983.

8. Topor R.W., "Termination Selection for Distributed Computing," Information Processing Letters, Vol.18, No.1, 1984, pp. 33-36.