© 2015 IJSRSET | Volume 1 | Issue 4 | Print ISSN : 2395-1990 | Online ISSN : 2394-4099
Themed Section: Engineering and Technology

An Iterative Method in Asynchronous System using Asynchronous Algorithms

Ajitesh S. Baghel, Rakesh Kumar Katare

Department of Computer Science, A. P. S. Univesity, Rewa Madhya Pradesh, India

ABSTRACT

In this paper, we exhibit a couple of established iterative routines for tackling straight comparisons; such systems are broadly utilized, particularly for the arrangement of vast issues, for example, those emerging from the discretization of direct fractional differential mathematical statements. We depict the iterative or aberrant strategies, which begin from a close estimation to the genuine arrangement and if joined, determine a grouping of close estimates the cycle of reckonings being rehashed till the obliged exactness is gotten. It implies that in iterative routines the measure of calculations relies on upon the exactness needed.

Keywords: Iteraitive, Ashynchornous Algorithm

I. INTRODUCTION

Asynchronous iterations have been introduced by Chazan and Miranker (1969) [5] for the solution of linear equation. The communication penalty, and the overall execution time of many algorithms, can often be substantially reduced by means of asynchronous implementations.[4] The main characteristic of an asynchronous algorithm is that the local Algorithms do not have to wait at predetermined points for predetermined messages to become available. We thus allow some processors to compute faster and execute more iterations than others, we allow some processors to communicate more frequently than others, and we allow the communication delays to be substantial and unpredictable. We also allow the communication channels to deliver message out of orders, that is, in a different order than the one in which they were transmitted.

Given a distributed algorithm, for each processor, there is a set of times at which the processor executes some computations, some other times at which the processor sends some messages to other processors, and yet some other times at which the processor receives messages from other processors. The algorithm is termed synchronous, in the sense of the Preceding subsection, if it is mathematically equivalent to one for which the times of computation, message transmission, and message reception are fixed and given a priori. We say that the algorithm is asynchronous if these times can vary widely in two different executions of the algorithm with an attendant effect on the results of the computation [4]. The most extreme type of asynchronous algorithm is one that can tolerate changes in the problem data or in the distributed computing system, without restarting itself to some predetermined initials conditions. Iterative methods, also known as trial and error methods, are based on the ideas of successive approximation. They start with one or more initial approximation to the root and obtain a sequence of approximations by repeating a fixed sequence of steps till the solution with reasonable accuracy is obtained. Iterative methods, generally, give one root at a time. Iterative methods are very cumbersome and time-consuming for solving non-linear equations manually. However, they are best suited for use on computers, due to following reasons:

1. Iterative methods can be concisely expressed as computational algorithms.
2. It is possible to formulate, using trial and error, algorithms which tackle a class of similar problems. For instance a general computational algorithm to

solve polynomial equations of order n(where n is an integer) may be written.
3. Routing errors are negligible in trial and error procedures compared to procedures based on closed form solutions.

In computational mathematics an iterative method is a mathematical procedure that generates a sequence of improving approximate solutions for a class of problems. A specific implementation of an iterative method, including the termination criteria, is an algorithm of the iterative method. An iterative method is called convergent if the corresponding sequence converges for given initial approximations. A mathematically rigorous convergence analysis of an iterative method is usually performed; however, heuristic-based iterative methods are also common.

A method uses iteration if it yields successive approximations to a required value by repetition of a certain procedure. An "iterative" process can be explained by the flowchart given in Fig. 2. There are four parts in the process, namely, initialization, decision, computation and update. The functions of the four parts are as follows:

1. Initialization: The parameters of the function and a decision parameter in this part are set to their initial values. The decision parameter is used to determine when to exit from the loop.
2. Computation: The required computation is performed in this part.
3. Decision: The decision parameter is to determine whether to remain in the loop.
4. Update: The decision parameter is updated, and a transfer to the next iteration results.

II. METHODS AND MATERIAL

A. Analysis of Parallel Algorithms

The most important three criteria we use to analyze a parallel algorithm are: running time, the number of processors in the computational model and cost [21]. The running time of a parallel algorithm is defined as the time taken by this algorithm to solve a problem on a parallel computer. Specifically, we are interested in the worst time, which means the time required for solving the most difficult instance of the problem using this algorithm. Usually, we count how many elementary steps are performed by an algorithm when solving a problem (worst case) as a measure of running time. There are two different kinds of elementary steps:

1. Computational steps: A computational step is an arithmetic or logic operation performed on a datum within a processor, like adding two numbers.
2. Routing steps: A routing step takes place when a datum of constant size is transmitted from one processor to another processor via shared memory or an interconnection network.

Each step (computational or routing) takes a constant number of time units, and the running time of a parallel algorithm is a function of the size of input. For a problem of size N, we use t (N) to denote the worst case number of time units required by the parallel algorithm. We use p(N) to denote the number of processors used by a parallel algorithm to solve a problem of size N And the cost c(N) of a parallel algorithm for a problem of size N is then defined as $c(N) = t(N) \times p(N)$. The cost of a parallel algorithm is an upper bound on the total number of elementary steps executed.

B. Overview of Asynchronous Algorithm

This situation arises principally in data networks, where the nodes and the communication links can fail or be repaired as various distributed algorithms that control the network are executed. In an asynchronous implementation of the iteration $x := f(x)$, processors are not required to wait to receive all messages generated during the previous iteration. Rather, each processor is allowed to keep iterating on its own component at its own pace. If the current value of the component updated by some other processors is not available, then some outdated value received at some time in the past is used instead. Furthermore processors are not required to communicate their results after each iteration but only once in a while. We allow some processors to communicate more frequently than others, and we allow the communication delays to be substantial and unpredictable. We also allow the communication channels to deliver messages out of order, i.e.; in a different order than the one they were transmitted.

C. Advantages of Asynchronous Algorithm

There are several potential advantages that may be gained from asynchronous execution [12]

1. Reduction of the synchronization penalty

There is no overhead such as the one associated with the global synchronization method. In particular, a processor can proceed with the next iteration without waiting for all other processors to complete the current iteration, and without waiting for a synchronization algorithm to execute. Furthermore, in certain cases, there are even advantages over the local synchronization method as we now discuss. Suppose that an algorithm happens to be such that each iteration leaves the value of xi unchanged. With local synchronization, processor i must still send messages to every processor j with (i, j) $\in$ A because processor j will not otherwise proceed to the next iteration. Consider now a somewhat more realistic case where the algorithm is such that a typical iteration is very likely to leave xi unchanged. Then each processor j with (i, j) $\in$ A will be often found in a situation where it waits for rather uninformative messages stating that the value of xi has not changed.

In an asynchronous execution, processor j does not wait for messages from processor i and the progress of the algorithm is likely to be faster. A similar argument can be made for the case where xi changes only slightly between iterations. Notice that the situation is similar to the case of synchronization via rollback, except that in an asynchronous algorithm processors do not roll back even if they iterate on the basis of outdated and later invalidated information.

2. Ease of Restarting

Suppose that the processors are engaged in the solution of an optimization problem and that suddenly one of the parameters of the problem changes. (Such a situation is common and natural in the context of data networks or in the quasistatic control of large scale systems.)
In a synchronous execution, all processors should be informed, abort the computation, and then reinitiate (in a synchronized manner) the algorithm.

In an asynchronous implementation no such reinitialization is required. Rather, each processor incorporates the new parameter value in its iterations as soon as it learns the new value, without waiting for all processors to become aware of the parameter change. When all processors learn the new parameter value, the algorithm becomes the correct (asynchronous) iteration.

3. Reduction of the Effects of bottlenecks:

Suppose that the computational power of processor i suddenly deteriorate drastically. In a synchronous execution the entire algorithm would be slowed down. In an asynchronous execution, however, only the progress of x_i and of the components strongly influenced by x_i would be affected; the remaining components would still retain the capacity of making unhampered progress. Thus the effects of temporary malfunctions tend to be localized. The same argument applies to the case where a particular communication channel is suddenly slowed down.

4. Convergence acceleration due to a Gauss-Seidel Effect

With a Gauss-Seidel execution, convergence often takes place with fewer updates of each component; the reason being that new information is incorporated faster in the update formulas. On the other hand Gauss-Seidel iterations are generally less parallelizable. Asynchronous algorithms have the potential of displaying a Gauss-Seidel effect because newest information is incorporated into the computations as soon as it becomes available, while retaining maximal parallelism as in Jacobi type algorithms.

D. Drawback of asynchronous algorithm

A major potential drawback of asynchronous algorithms is that they cannot be described mathematically by the iteration x (t + 1) = f (x (t)). Thus, even if this iteration is convergent, corresponding the asynchronous iteration could be divergent, and indeed this is sometimes the case. Even if the convergence of the asynchronous iteration can be established, the corresponding analysis is often difficult.

Another difficulty relates to the fact that an asynchronous algorithm may have converged (within a desired accuracy) but the algorithm does not terminate because no processor is awrare of this fact.

- An interesting fact is that some asynchronous algorithms, called totally asynchronous, or chaotic, can tolerate arbitrarily large communication and computation delays, while other asynchronous algorithms, called partially asynchronous are not guaranteed to work unless there is an upper bound on these delays. The convergence mechanisms at work in each of these two cases are genuinely different and so are their analyses.

Here we discuss the model of asynchronous computation [4]. Let the set X and the function f. Let t be an integer variable used to index the events of interest in the computing system. Although t will be referred to as a time variable, it may have little relation with "real time". Let $x_i(t)$ be the value of x_i residing in the memory of the i^{th} processor at time t. We assume that there is a set of times T_i at which x_i is updated. To account for the possibility that the i^{th} processor may not have access to the most recent values of the components of x, we assume that

$$x_i(t+1) = f_i(x_1(r_i^1(t)),,..x_n(r_n^i(t))), \forall t \in T^i \quad (1.1)$$

Where $T_J^i(t)$ are times satisfying

$0 < t_j^i(t) \leq t, \ \forall t \geq 0.$

At all times $t \notin T^i$, $x_i(t)$ is left unchanged and

$$x_i(t+1) = x_i(t), \ \forall t \in T^i \quad (1.2)$$

We assume that the algorithm is initialized with some $x(0) \in X$.

The above mathematical description can be used as a model of asynchronous iterations executed by either a message passing distributed system or a shared memory parallel computer.

The difference $t - t_j^i(t)$ is equal to zero for a synchronous execution. The larger this difference is, the larger is the amount of a synchronism in the algorithm. Of course, for the algorithm to make any progress at all we should not allow $t_j^i(t)$ to remain forever small. Furthermore, no processor should be allowed to drop out of the computation and stop iterating. For this reason, certain assumptions need to be imposed. There are two different types of assumptions which we state below.

Assumption1.1 (Total asynchronism) The sets Ti are infinite and if $\{t_k\}$ is a sequence of elements of T^i which tends to infinity, then $\lim_{k\to\infty} t_j^i(t_k) = \infty$ for every j.

Assumption 1.2 (Partial asynchronism) There exists a positive constant B such that:

(a) For every $t \geq 0$ and every i, at least one of the elements of the sets $\{t, tl + 1,\ldots, t + B-1\}$ belongs to T^i.

(b) There holds

$$t - B < t_j^i(t) \leq t, \ \forall i,j, \quad \forall t \in T^i \quad (1.3)$$

(c) There holds $t_j^i(t) = t$, for all i and $t \in T^i$.

The constant B of Assumption 1.2. to be called the asynchronism measure, bounds the amount by which the information available to a processor can be outdated. Notice that a Jacobi-type synchronous iteration is the special case of partial asynchronism in which B=1.

Notice also that Assumption 1.2(c) states that the information available to processor i regarding its own component is never outdated. Such an assumption is natural in most contexts, but could be violated in certain types of shared memory parallel computing systems if we allow more than one processor to update the same component of x. It turns out that if we relax Assumption 1.2(c), the convergence of certain asynchronous algorithm is destroyed [14, 4]. Parts (a) and (b) of Assumption 1.2 are typically satisfied in practice.

Asynchronous algorithm can exhibit three different types of behavior (other than guaranteed divergence):

a) Convergence under total asynchronism.
b) Convergence under partial asynchronism, for every value of B, but possible divergence under totally asynchronous execution.
c) Convergence under partial asynchronism if B is small enough and possible divergence if B is large enough.

Tottaly Asynchronous Algorithm

Totally asynchronous convergence results have been obtained by Chazan and Miranker (1969) [5] for linear iterations, Miellou (1975a) [17], Baudet (1978) [1], EI Tarazi (1982) [9], Miellou and Spiteri (1985) [19] for contracting iterations, Miellou (1975b) [18] and Bertsekas (1982) [2] for monotone iterations, and Bertsekas (1983) [3] for general iterations. Related results can be also found in [23, 24, 25]. The following general result is from (Bertsekas, 1983) [3].

Proposition 1.1. Let $X = \Pi^{p}_{I=1} \subset \Pi^{p}_{i=i} R^{ni}$. Suppose that for each i E {1, . . ,p}, there exists a sequence {X,(k)} of subsets of X_i such that:

(a) $X_i(k + 1) \subset x_i (k)$ for all $k > 0$.

(b) The sets $X(k) = \Pi^{p}_{i=i}X_i(k)$ have the property $f(x) \in X(k + 1)$

(c) Every limit point of a sequence {x(k)} with the property $x(k) \in X(k)$ for all k, is a fixed point of f

Then, under Assumption 1.1 (total asynchronism), and if $x(0) \in X(0)$, every limit point of the

Sequence {x (t)} generated by the asynchronous iteration (1.1)-(1.2) is a fixed point of f

Proof: We show by induction that for each $k > 0$, there is a time t_k such that:

(a) $x(t) \in X(k)$ for all $t \geq t_k$.

(b) For all i and $t \in T^i$ with $t \geq t_k$, we have $x^i \in x(k)$ where

$x^i = (x_l (t_j^i (t)), x_z (t, (t)), x_n (t, (t))), \quad \forall t \in T^i$

E. Partially Asynchronous Algorithms

We now consider iterations satisfying the partial asynchronism Assumption 1.2. Since old information is "purged" from the algorithm after at most B units, it is natural to describe the "state of the algorithm at time t by the vector $z(t) \in XB$ defined by

$z(t) = (x(t), x(t - 1), ..., x(t - B + 1))$

We the n notice that x(t + 1) can be determined [cf. Eqs. (1.1)-(1.3)] in terms of z(t); in particular k, knowledge of x(r), for $r \leq t - B$ is not needed. We assume that the iteration mapping f is continuous and has a nonempty set $X^* \subset X$ of fixed points. Let Z* be the set of all vectors $z^* \in X^B$ of the form $z^* = (x^*, x^*,..., x^*)$, where x* belongs to X*. We present a sometime useful convergence result, which employs a Lyapunov-type function d defined on the set X^B.

F. Termination Of Asynchronous Iterations:

In practice iterative algorithms are executed only for a finite number of iterations, until some termination condition is satisfied. In the case of asynchronous iterations, the problem of determining whether termination conditions are satisfied is rather difficult because each processor possesses only partial information on the progress of the algorithm.

We now introduce one possible approach for handling the termination problem for asynchronous iterations. In this approach, the problem is decomposed into two parts:

a) An asynchronous iterative algorithm is modified so that it terminates in finite time.
b) A special procedure is used to detect termination in finite time after it has occurred.

In order to handle the termination problem, we have to be a little more specific about the model of interprocessor communication. While the general model of asynchronous iterations introduced in Section 5 can be used for both shared memory and message-passing parallel architectures, we adopt here a more explicit message-passing model. In particular, we assume that each processor j sends messages with the value of x_j to every other processor i. Processor i keeps a buffer with the most recently received value of x_j We denote the value in this buffer at time t by x_j^i . this value was transmitted by processor j at some earlier time t_j^i (t) and therefore x_j^i (t) = x_j (t_j^i) (t)). We also assume the following:

Assumption1.3

a) If $t \in T^i$ and $x_i (t + 1) \neq x_i (t)$ then processor i will eventually send a message to every other processor.
b) If a processor i has sent a message with the value of x_i (t) to some other processor j, then processor i will send a new message to processor j only after the value of x_i changes (due to an update by processor i).
c) Messages are received in the order that they are transmitted.
d) Each processor sends at least one message to every other processor.

Assumption 1.3(d) is only needed to get the algorithm started. Assumption 1.3(b) is crucial and has the following consequences. If the value of x(t) settles to some final value, then there will be some time t*after which no messages will be sent. Furthermore, all messages transmitted before t* will eventually reach their destinations and the algorithm will eventually reach a quiescent state where none of the variables x_i changes and no message are in transit. We can then say that the algorithm has terminated.

More finally, we view termination as equivalent to the following two properties

(i) No, message is in transit.
(ii) An update by some processor i cause no change in the value of x_i.

Property (ii) is a collection of local termination conditions. There are several algorithms for termination detection when a termination condition can be decomposed as above [6, 4]. Thus termination detection causes no essential difficulties, under the assumption that the asynchronous algorithm terminates in finite time.

III. CONCLUSION

In this we examine circulated calculation, for every processor, there is a situated of times at which the processor executes a few processing's, some different times at which the processor sends a few messages to different processors, but some different times at which the processor gets messages from different processors. The calculation is termed synchronous, in the feeling of the former subsection, on the off chance that it is scientifically proportional to one for which the seasons of reckoning, message transmission, and message gathering are altered and given from the earlier. We say that the calculation is non-concurring if these circumstances can shift generally in two distinct executions of the calculation with a chaperon impact on the aftereffects of the reckoning.

IV. REFERENCES

[1] Baudet, G.M. (1978). Asynchronous iterative methods for multiprocessors, Journal of the ACM, 2, pp. 226-244.

[2] Bertsekas, D.P. (1982). Distributed dynamic programming, IEEE Transactions on Automatic Control, AC-27, pp. 610-616.

[3] Bertsekas, D.P. (1983). Distributed asynchronous computation of fixed points, Mathematical Programming, 27, pp. 107-120.

[4] Bertsekas, D.P., and J.N. Tsitsiklis (1989). Parallel and Distributed Computation: Numerical Methods, Prentice Hall, Englewood Cliffs, NJ.

[5] Chazan, D., and W. Miranker (1969). Chaotic relaxation, Linear Algebra and its Applications, 2, pp. 199-222.

[6] Dijkstra, E.W., and C.S. Scholten (1980). Termination detection for diffusing computations, Information Processing Letters, 11, pp. 1-4.

[7] Dubois, M., and F.A. Briggs (1982). Performance of synchronized iterative processes in multiprocessor systems, IEEE Transactions on Software Engineering, 8, pp. 419-431.

[8] E.D. Dekel , Nassimi , S. Sabni. "Parallel matrix and graph algorithms," SIAM J. Comput., 10(4),657-675, 1981.

[9] El Tarazi, M.N. (1982). Some convergence results for asynchronous algorithms, Numerisch Mathematik, 39, pp. 325-340.

[10] Fortune,S; and J. Wyllie. 1978 Parallelelism in random access machines, proceedings of the 10thAnnual ACM Symposium on theory of computing, PP, 114-118

[11] J. Ammon. "Hypercube Connectivity within cc NUMA architectore, Silicon Graphics, 20 ILN," Shoreline Blvd. Ms 565, Mountain View, CA94043.

[12] Kung, H.T. (1976). Synchronized and asynchronous parallel algorithms for multiprocessors, in Algorithms and Complexity, J.F. Traub (Ed.), Academic, pp. 153-200.

[13] Lavenberg, S., R. Muntz, and B. Samadi (1983). Performance analysis of a rollback method for distributed simulation, in Performance 83, A.K. Agrawala and S.K. Tripathi (Eds.), North Holland, pp. 117-132.

[14] Lubachevsky, B., and D. Mitra (1986). A chaotic asynchronous algorithm for computing the fixed

point of a nonnegative matrix of unit spectral radius, Journal of the ACM, 33, pp. 130-150.

[15] M. Flynn. Some computer organizations and their effectives. IEEE Trans. Comput., C(21):948-960, 1972.

[16] M.J. Quinn. "Parallel Computing," Mc Graw-Hill, INC, 1994.

[17] Miellou, J.C. (1975a). Algorithmes de relaxation chaotique a retards, R.A.IR.O., 9, R-1, pp. 55-82.

[18] Miellou, lC. (1975b). Iterations chaotiques a retards, etudes de la convergence dans Ie cas d'espaces partiellement ordonnes, Comptes Rendus, Academie de Sciences de Paris, 280, Serie A, pp. 233-236.

[19] Miellou, J.C., and P. Spiteri (1985). Un critere de convergences pour des methods generales de point fixe, Mathematical Modelling and Numerical Analysis, 19, pp. 645-669.

[20] Mitra, D., and I. Mitrani (1984). Analysis and optimum performance of two message passing parallel processors synchronized by rollback, in Performance '84, E. Gelenbe (Ed.), North Hol-Iand, pp. 35-50.

[21] S. G. Akl. The Design and Analysis of Parallel Algorithms. Prentice Hall, Englewood Cliffs, 1997.

[22] S. G. Ald. Parallel Computation: Models And Methods. Prentice Hall, Upper Saddle River, 1997.

[23] Uresin, A., and M. Dubois (1986). Generalized asynchronous iterations, in Lecture Q Notes in Computer Science, 237, Springer Verlag, pp. 272-278.

[24] Uresin, A., and M. Dubois (1988a). Sufficient conditions for the convergence of asynchronous iterations, Technical Report, Computer Research Institute, University of Southern California, Los Angeles, California, U.S.A.

[25] Uresin, A. and M. Dubois (1988b). Parallel asynchronous algorithms for discrete data, Technical Report CRI-88-05, Computer Research Institute, University of Southern California, Los Angeles, California, U.S.A.