1993

Clock synchronization in multiprocessor systems

Gokuldas K. Hegde
The University of Montana

CLOCK SYNCHRONIZATION IN MULTIPROCESSOR SYSTEMS

BY
GOKULDAS HEGDE K
B.E., University of Mysore--INDIA, 1984.

THESIS REPORT

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at Graduate School of University of Montana.

April, 1993

Approved by

[illegible]

Principle Advisor: Ray F. Ford
Associate Professor CS Dept

[illegible]

Dean, Graduate School

May 5, 1993

Date

gift 6-11-93

ABSTRACT

GOKULDAS HEGDE K., MS APRIL 1993 COMPUTER SCIENCE

CLOCK SYNCHRONIZATION IN MULTIPROCESSOR SYSTEMS

Principle Advisor: Ray F. Ford, Associate Professor CS Dept.

[illegible]

A fault tolerant algorithm to synchronize clocks in multiprocessor systems, which is independent of local clock conditions, and unrestricted in terms of minimal connectivity requirements is proposed. In the proposed algorithm each processor receives clock values from its adjacent processors and computes the error correction value after filtering out the faulty clock values in two stages. Usage of difference of clock values and filtering based on limits set at each stage, eliminates the drawbacks of previously proposed algorithms.

The algorithm is compared with two other algorithms for performance in terms of synchronization achieved and fault tolerance. A software simulation system written in Ada implements a multiprocessor system. Drifts can be introduced into each processor to simulate errors in clocks. The error correction values generated by the algorithms are compared for performance.

Results obtained by the simulation run demonstrate the advantages of the proposed algorithm in certain situations. The algorithm is independent of the local clock, and does not contain restrictions on the number of minimum connectivity required to be fault tolerant in terms of number of maximum faulty clocks tolerated by the algorithm before failing. The synchronization achieved is better or within tolerable limits in all cases.

ACKNOWLEDGEMENT

I express my heartfelt thanks for Dr. Ray Ford for his invaluable guidance, suggestions and comments in every phase of the work. I also thank Dr Youlu Zheng, and Prof. Paul B. Wilson for accepting to serve as members of advisory committee and their guidance. My thanks to the Chair of the Computer Science department for extending all the facilities and computers for carrying out this work. I also thank all the faculty and staff of the Computer Science department for their cooperation and assistance given to me through out my work.

GOKULDAS HEGDE K.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION.

One of the problems in a multiprocessor or distributed system is synchronization of multiple independent clocks. Every processor in a multiprocessor or distributed system contains its own clock. By nature these clocks have a tendency to drift away from their ideal time. This is due to varying frequencies of oscillators used in implementing the clocks. Clocks which are supposed to run at a definite rate may be slower or faster than the standard rate, thus creating differences between their actual and ideal value. Factors causing variations in the oscillator frequency are many, e.g., humidity, temperature, crystal oscillators, and electrical behavior of electronic components used in the oscillator circuitry. The problem of bounding the variation between a set of independent clocks, each varying to some extent from an ideal rate is referred to as the CLOCK SYNCHRONIZATION problem.

In designing multiprocessor and distributed systems the need often arises to generate a unique global clock, which can be referenced by all processors in the system. Clocks from different processors tend to drift at an unpredictable rate, thus a time value in one processor is generally not valid in another processor, whose local clock drifts at a different rate. This necessitates creation of a global clock standard to be referred by all the processors in the network of processors.

There are several methods that can be used to create a global clock. Factors such as the cost of implementation and relative accuracy generally determine the feasibility of any particular method for a given application. Apart from these, other factors that are of importance, are the tolerance of the synchronization method to faulty clock values and faulty processors or communication subsystem. In any case best approaches typically can only bound the drift error and some small difference in clock values must be tolerated by the system. The selection of any method for clock synchronization thus requires careful analysis of the methods cost and the accuracy that it can achieve.

The following chapter contains a review of relevant work in this field, along with a brief discussion about the general methods used to implement a global clock, and their advantages and disadvantages. Certain parameters and relevant terms involving clock synchronization are also described. Assumptions made in implementing clock synchronization in various methods are also explained.

In Chapter Three a novel method for implementing clock synchronization is proposed. Comparison is made with two existing algorithms which fall under the same category, in terms of implementation cost and abstract complexity.

Chapter Four contains the description of a simulation system built to study the performance of the three clock synchronization algorithms in various simple contexts. All the three methods under study are implemented and verified for results. Data obtained by simulating the performance is also shown and explained.

Chapter Five contains the conclusions drawn from analysis of the simulation data, along with possible additional modifications that could be implemented to improve the proposed algorithms.

CHAPTER 2

SURVEY OF RELEVANT WORK

Several methods have been proposed and implemented to synchronize clocks. A general survey of relevant algorithms of interest is presented in this chapter.

UPDATE METHODS

Attaining a common clock between all the processors can be achieved in many ways. A simple method described by Levi and Agarwala [5] is by establishing a MASTER and SLAVE relation between the processors. One of the processors is selected as the MASTER and all the other processors are SLAVES. The MASTER transmits the clock to the SLAVES and the SLAVES operate using this clock. This arrangement is only appropriate for tightly coupled processors because the transmission delay of clocks causes difference in clocks among the processors. Failure of the Master clock brings down the entire system. Hence this method is unsuitable in critical applications that require fault tolerance.

Another method described by Parameshwaran Ramanathan, Kang G. Shin, Ricky W. Butler [2], Anne Dinning [3], and Leslie Lamport and P. M. Melliar-Smith [6] involves generating a logical clock at each of the processors using the local clock and clock values from the adjacent processors. The logical clock value is computed based on an algorithm, which we call the clock synchronization algorithm. The logical clock can be generated by various methods: hardware, software, and hybrid, a combination of both hardware and software. Ramanathan, Shin, and Butler [2] briefly explain these methods. They refer to Hardware methods as "Continuous Update Methods" because the special hardware allows the clocks to be updated or corrected continuously in real time. The clock values are transmitted to adjacent processors continuously, and hardware-based algorithms, such as phase correction or frequency correction, are used to generate an error signal from the received clocks. They refer to software methods as "Discrete Update Methods", because the clock correction is calculated and updated in discrete intervals by an algorithm implemented in software. Software methods are further classified into three categories: convergence averaging algorithms, convergence nonaveraging algorithms, and consistency algorithms.

Generally all software methods involve transmitting a clock value from processor A to B at a certain time interval. The clock values thus received by B are used by B to compute the error value, and to generate the corrected global clock. Assumptions made in these methods are that the clocks do not drift beyond certain limits within the synchronization period, and that no two processors differ in their clock values by more than a certain limit at any time. Several different software algorithms have been proposed, two of which are discussed in more detail below.

Hybrid methods use a combination of both hardware and software techniques. The extent of hardware and software involved depends upon the method of synchronization.

REVIEW OF EXISTING ALGORITHMS

Any study of clock synchronization algorithms assumes that the global clock is used to provide critical software synchronization for distributed systems. Lamport [1] describes the concepts of "time" and "clocks" in distributed processing along with the importance of clock synchronization and global clock based event ordering.

All global clock synchronization algorithms must satisfy two key conditions, though the precise structure of the conditions are suitably stated or modified for various methods of synchronization. The basic form of these conditions is stated below. Consider N processors, and let P_r,P_a, and P_b be any three processors:

1. **Any two nonfaulty processes P_a and P_b obtain approximately the same value for P_r's clock even if P_r is faulty.**

2. **If P_r is nonfaulty then every nonfaulty processor obtains approximately the same value of P_r's clock.**

Apart from satisfying these two conditions several other factors affect the clocks. Parameters like transmission delays incurred while conveying time values from one processor to another, time required for reading clocks, and time for computing the clock synchronization algorithm, are of importance in implementing a synchronization system. Several assumptions are made about these and other time intervals while designing an algorithm. The time elapse between two synchronization events, which is called the "**Synchronization period**", plays a major role in considering various delays. For larger synchronization periods, time intervals such as transmission and clock reading delays may be neglected. Several articles [1,2,6,7] have detailed analysis of such assumptions and proofs.

As described in [2] and [3] synchronization algorithms can be broadly classified as Interactive Convergence Algorithms and Interactive Consistency Algorithms. One example coming under the classification of Interactive Convergence Algorithms is the algorithm explained in [3,7], which we refer to as CNV. In this algorithm the clock synchronization correction value is computed as the average of the clock values read by the processor. This algorithm is one of the algorithms used in analysis of the proposed algorithm for comparison purposes.

The other two algorithms described in [3,7], COM and CSM are classified under Interactive Consistency Algorithms. In algorithm COM the correction value is computed as the "median" of the clock values received from the adjacent processors. Algorithm CSM associates a "signature" from every processor handling the clock value and calculates the correction value from the signatures and incoming clock values. In this scheme it is assumed that every processor generates a unique signature that cannot be altered by other processors. Every processor attaches its signature with every clock it reads and transmits. Each processor verifies the signatures associated with the clock it receives before validating the clock. By this method each processor ascertains that the clock is read by m+1 processors by identifying m+1 signatures, where m is the number of faulty clocks tolerated in the system. This is to assert that at least one nonfaulty processor has read the clock. This satisfies a modified requirement of the conditions stated above. The discussion in [2,3] also describes problems associated with reading clock values and methods to reduce errors in reading clocks.

Further classification of Interactive Convergence algorithms leads to the two sub-categories Convergence Averaging and Convergence Nonaveraging algorithms. In Convergence Averaging Algorithms each processor computes a fault tolerant average from the clock values it receives from the adjacent processor clocks. This is similar to the Interactive Convergence Algorithm. In Convergence NonAveraging Algorithms not all processors compute a correction value. This synchronization process follows a MASTER-SLAVE relation in which one processor computes the correction value and behaves as a system synchronizer, sending the correction value to all other processors (i.e., slaves). The processors may also take turns in acting as the system synchronizers.

Schemes for synchronizing networks are explained in [3]. Synchronization can be achieved at different levels. At a lower level a few nodes connected as a cluster can be locally synchronized. Several such clusters can be interconnected and synchronized to form higher levels of synchronization. Hybrid synchronization methods are often used in such applications to avoid high cost of hardware synchronization yet achieve tighter synchronization than which can be achieved through software only synchronization methods. Dinning [4] introduces several synchronization mechanisms used in different parallel computers and discusses implementations of synchronization like semaphore, monitors and message passing methods. A formal explanation of such methods is given by Welch and Lynch [6]. Generally in these applications, if m is the number of faulty processors tolerated, then each processor on receiving clocks from its adjacent nodes rejects the highest m and lowest m values and then computes the correction value. The second algorithm to be used as a basis for comparing our proposed algorithm is one derived from the Welch and Lynch proposal. We refer to this algorithm as LW for analysis and comparison purposes.

The requirements for an algorithm to be fault tolerant, as explained in [5] are,

> "In a Comprehensive approach to constructing a fault-tolerant time-server, one must start with providing the means for a local resynchronization of each time server in the system. This resynchronization updates the server's parameters, and thus the interpretation of the local clock. However, one must introduce additional facilities such as clock broadcasts and participant-forum establishment. This additional support serves the requirements for fault tolerance. Combining the above facilities results in a comprehensive solution to a system-wide distributed time service".

Hence any system built to be fault tolerant must have facilities for local resynchronization, clock broadcast, and a forum for participant processors to communicate.

REFERENCE ALGORITHMS

The two algorithms falling under the same category of Interactive Convergence Averaging Algorithms are selected as reference algorithms to compare with our proposed algorithm. These algorithms are simple in terms of the fault tolerant averaging function used to compute the correction. Thus do not involve complications of generating complex signatures for each processor in the network, as required by other class of algorithms. Finally, these algorithms do not involve extensive or specialized communication protocols. The two algorithms are the CNV and LW algorithms described above. More details on the methods used in these two algorithms are presented below.

Algorithm LW, requires that m maximum and minimum values be eliminated. This imposes a condition on the number of processors that must be connected to any processor. Since 2 * m values are eliminated, and there must be at least one value for computation of the correcting value, the minimum number of processors that must be connected to any processor P_r is,

<1> $\text{Min\_connectivity}(P_r) = (2 * m) + 1$

Algorithm CNV uses clock values from its adjacent processors to compute its clock correction value. Though no limitations exist on number of processors that must be connected, the process of validating clock values is of concern. Each clock value received from an adjacent processor is compared with the local clock value. If this difference exceeds a specified limit the adjacent processor's clock value is eliminated from computation. The valid clock values are used to compute the average which is used as the correction value. This causes problems in case the local clock is faulty. A faulty local clock would qualify clocks which are out of range, thus skewing the correction value towards the faulty clocks. The cumulative effect pushes the processor further out of synchronization, rather than pushing it into synchronization. This also makes it hard to reintroduce a repaired processor into the network, without the network being halted.

CHAPTER 3

PROPOSED ALGORITHM

PROPOSAL

This thesis proposes an algorithm that overcomes some of the drawbacks of the two algorithms mentioned in the previous chapter. The proposed algorithm herein after referred as algorithm " **GH** ", can be compared with these two algorithms in terms of the synchronization achieved and cost of implementation. A theoretical comparison can be made for the worst case cost of computation, relative to a particular network and particular node. Comparison can also be based on performance.

In most of the above mentioned algorithms one assumption made about the processors is that every processor receives clock values from every other processor and then computes the error value. Certainly, this may not be necessary based on the transitive nature of the problem. Consider three nodes A, B and C connected as shown in Figure 1. If node B is synchronized with node A, and node C is synchronized with node B, then certainly node C is

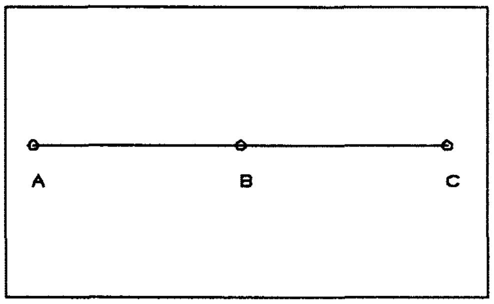

Figure 3.1

synchronized with respect to A. Thus by transitivity, it may not be necessary for every node in a system to be synchronized with every other node.

In the new GH algorithm the processor under consideration reads clock values from upto N neighboring connected processors. Each processor then computes the average difference between its local clock and the clock values it receives from its neighbors. A filtering is done with the average value as the midpoint, and using a bandwidth parameter that depends upon the accuracy of the synchronization required. The bandwidth is the range up to which the values are accepted both on the higher and lower side of the midpoint value. Filtering consists of marking all those difference values exceeding the specified bandwidth and calculating a new average based upon those values falling within the bandwidth. The filtering reduces the skew created by out of range values and helps obtain a more accurate correction value. This filtering is done at two stages to reduce the errors. At each stage the filtering marks out of range clock values as "bad" and helps reduce the effects of faulty neighboring processors.

The following section contains the details and comparison of three algorithms: the newly proposed GH algorithm, the interactive consistence algorithm CNV and the fault tolerant algorithm LW. For each algorithm we show the algorithm (in pseudocode), present a brief explanation of each of its stages of computation, and finally describe its abstract time complexity.

LIST OF SYMBOLS

A list of the symbols used in the pseudocode and analysis is given below.

DEG_i :	Number of processors connected to processor i.
CLK(J):	Clock value read from adjacent processor J. This is an array to store all the clock values read from adjacent processors.
LMT1 :	Range limit for qualifying values for first level filtering in GH.
LMT2 :	Range limit for qualifying values for second level filtering in GH.
D_LMT :	Range limit for qualifying values in the CNV algorithm.
AVG_i :	Averages computed from the values.
T_r :	Time on average for clock read.
T_{add} :	Time to compute an average.
T_{comp} :	Time for comparison.
T_{avg} :	Time for division or computation of average.
T_{stagei} :	Time for a certain stage 'i' of computation.

T_{total} : Total time for algorithm.

PSEUDOCODE:

```
# Read $DEG_i$ clock values; N1 = $DEG_i$
1 > for INDX1 from 1  to $DEG_i$  do
2 >   " Read CLK( INDX1 ) and compute SUM "
3 > end for;
4 > AVG1 := SUM / $DEG_i$;
# stage 1 filtering and computation.
5 > for INDX1 from 1  to $DEG_i$  do
6 >     " Filter based on AVG1 and compute SUM
        for remaining N2 filtered clock values ".
7 > end for;
8 > AVG2 := SUM / N2;
# stage 2 filtering and computation.
9 > for INDX1 from 1  to N2 do
10>     " Filter based on AVG2, and compute SUM
         for remaining N3 filtered clock values ".
11> end for;
12> CORR := SUM / N3.
```

Figure 3.2: ALGORITHM GH

GH DEFINITION AND ANALYSIS

The GH Algorithm is presented in Figure 3.2. Lines 1 to 4 constitute the clock read, sum, and average computation. Assuming N1 = Deg_i, the time computation for this stage is

$$< 2 > \qquad T_{stage1} := N1 * T_r + T_{ave}(N1)$$

Lines 5 to 8 are the first filter stage. In this step the clocks are compared and all values passing the comparison test are used to compute the sum for next stage of computation. Assuming N2 is the number of clock values qualifying from the comparison operation and satisfies condition $N2 \le N1$, the computation time for this stage is

< 3 > $$T_{stage2} := N2 * T_{comp} + T_{ave}(N2)$$

Lines 9 to 12 are the second filtering stage. Following filtering the number of output values N3 satisfies $N3 \le N2$. The computation time is

< 4 > $$T_{stage3} := N3 * T_{comp} + T_{ave}(N3)$$

The total time for execution of this algorithm is given by the sum of execution time for three stages.

< 5 > $$T_{total} := T_{stage1} + T_{stage2} + T_{stage3}$$

< 6 > $$T_{total} := N1*T_r + N2*T_{comp} + N3*T_{comp} + T_{ave}(N3) + T_{ave}(N2) + T_{ave}(N1)$$

< 7 > $$T_{total} := N1*T_r + T_{ave}(N1) + T_{ave}(N2) + T_{ave}(N3) + (N2+N3)*T_{comp}$$

Since $N3 \le N2 \le N1$, we replace N2 and N3 by N1 and bound the total exec time as

< 8 > $T_{total} \leq N1*T_r + 3*T_{ave}(N1) + 2*N1*T_{comp}$

PSEUDOCODE:

```
# Read N1 = $DEG_i$ number of clock values, filter, and sum.
1 > for INDX1 from 1  to $DEG_i$  do
2 >      " Read CLK( INDX1 ), filter based on local clock
value, leaving N2 clock values, and compute sum ".
3 > end for;
4 > CORR := SUM / N2.
```

Figure 3.3: ALGORITHM CNV

CNV DEFINITION AND ANALYSIS

The details of CNV are shown in Figure 3.3. Lines 1 to 3 implement clock reads, validation and summation of clock values. The validation is done by comparing each clock value to some parameter giving the limit for the maximum deviation allowed. The computation time is

< 9 > $T_{stage1} := N1*T_r + N1*T_{comp} + T_{ave}(N1)$

PSEUDOCODE:

```
# Read N1 = $DEG_i$ number of clock values.
1 > for INDX1 from 1  to $DEG_i$  do
2 >    " Read CLK( INDX1 ) ".
3 >    " Find m maximum and m minimum values ".
4 > end for;
5 > Delete m maximum and m minimum values.
6 > for INDX1 in 1 to N1 - 2*m do
7 >     " sum the values "
8 > end for;
9 > CORR := SUM / ( N1 - 2 * m ).
```

FIGURE 3.4: ALGORITHM LW

LW DEFINITION AND ANALYSIS

The details of LW are shown in Figure 3.4. Lines 1 to 4 implement clock reads and identify m maximum and m minimum values. The time for finding m maximum and minimum values is estimated as m * N1 * T_{comp}. The time for the read stage is as

< 10 > $T_{stage1} := N1 * T_r + m * N1 * T_{comp}$

Line 5 is the deletion stage. Lines 6 to 9 constitute the summation and computation of correction value. The computation time and total time are shown below.

< 11 > $T_{stage2} := T_{ave}(N1 - m)$

< 12 > $T_{total} := T_{stage1} + T_{stage2}$

< 13 > $T_{total} := N1*T_r + m*N1*T_{comp} + T_{ave}(N1 - 2*m)$

COMPARATIVE ANALYSIS:

I . COMPLEXITY ANALYSIS:

Comparison of the total time for the three algorithms gives a relative measure of the potential cost for the three algorithms. The three equations are given below.

GH: $T_{total} := N1*T_r + 3*T_{ave}(N1) + 2*N1*T_{comp}$

CNV: $T_{total} := N1*T_r + N1*T_{comp} + T_{ave}(N1)$

LW: $T_{total} := N1*T_r + m*N1*T_{comp} + T_{ave}(N1 - 2*m)$

Assuming that N1 is the same for all the three cases, we compare the computation cost for the three cases. Canceling the common factor in all the three cases i.e., $N1 * T_r$, we compare the relative cost, R_{cost}, of remaining factors. Let the cost of computation of $T_{comp} + T_{ave}$ be equal to some value " k ". Considering the worst case we have the cost of computation for each case.

GH: $R_{cost} := 3 * N1 * k$

CNV: $R_{cost} := N1 * k$

LW: $R_{cost} := m * N1 * k$

From the above three equations we note that the computation cost for CNV is the least, and that the cost for LW is less than that of GH only under the condition of $m < 3$. Also, note that the cost of GH is a constant irrespective of the degree of fault tolerance, but the cost for LW increases with the degree of fault tolerance.

In reality the value of N1 varies for the three algorithms. In CNV the value of N1 is assumed to be the total number of processors in the network, i.e., $N1 = N$. In LW the value of N1 has to satisfy the condition given by < 1 >, but it is assumed to be less then the total number of processors in the network. Substituting f for m in < 1 >, then if f is one then N1 cannot be less than 3, so that $3 \leq N1 < N$. For algorithm GH no apriori constraints exist on N1, i.e., $1 \leq N1 \leq N$.

Certain drawbacks of the algorithms are discussed below. In CNV the incoming clock values are compared with the local clock value and qualified for computation. This would cause serious errors if the local clock is erroneous. In LW and GH the local clock is used only to compute the clock difference. In case of LW the error correction value is computed by eliminating m maximum and m minimum values.

In the case of GH the error value is computed based on computing the average of the clock differences and selecting clocks lying within suitable range. Since the local clock is not the only basis for computation of the error correction value, both the algorithms LW and GH are more suitable to tolerate local clock skew.

Algorithm LW imposes a condition on the minimum connectivity required for any processor, based on the degree of fault tolerance specified. Hence if the network is to tolerate upto one faulty processor then the minimum degree of connectivity of any processor has to be three by equation < 1 >. This does not fully address problems such as the inability of a processor P_A to read the adjacent processor P_B's clock. The inability to read an adjacent processor's clock within a specified timeout period reduces the number of clock values available for computation. Allowing for such failures implies that actual connectivity must be greater than the minimum connectivity given in equation <1>.

GH is designed to overcome the drawbacks described above, i.e., to reduce errors in computation due to erroneous local clock, and to reduce the minimum connectivity required for fault tolerance. The table below gives a comparative fault tolerance for each algorithm.

ALGORITHM:	**GH**	**CNV**	**LW**
CONNECTIVITY:	N	N	$2 * m + 1 + N_{to}$[1]
FAULT-TOLERANCE:	N-1	N-1	m

This comparison shows that algorithm GH and CNV are better in terms of fault tolerance than LW. Hence for the same degree of fault tolerance the connectivity required for algorithm GH and CNV is less than that for algorithm LW.

Finally comparing the cost of computation for the three algorithms we have:

ALGORITHM:	GH	CNV	LW
COST:	3*N*k	N*k	m*N*k

Cost comparison shows that for small values of tolerance m cost for algorithm LW is low. But since N is large in case of LW the cost is higher than the other two algorithms. The value of N is relatively small in case of GH and CNV, thus reducing the cost considerably.

Thus, based on this high level comparison algorithm GH provides better fault tolerance than LW and CNV with a reasonable constant-bounded increase in compute cost.

[1] N_{to} = number of values unable to read beyond time-out.

CHAPTER 4

SIMULATION SYSTEM

SYSTEM REQUIREMENTS

The simulation system requirements are

1. A network of multiple processors.
2. Communication medium and means for communicating clocks.
3. A process or means to simulate simultaneous startup of the simulation system, thus satisfying the assumptions made about the algorithms.
4. A process or means to simulate clock drift.

Building a simulation system using hardware components would require multiple processors, a communication network, and extensive sophisticated monitoring hardware and software. Building a software simulation systems requires that comparable facilities be built in software. A hardware based simulator would be expensive and somewhat limited in scope. Software methods on the other hand are much cheaper and flexible. Any programming language which has the means to implement multiple processes would be a suitable choice to implement a simulation system. One such language is Ada, which directly supports multiple processes through its task construct and interprocess communication facilities based on "accept" statements and "entry" calls. Ada is the language used to implement the simulation system described here.

SYSTEM DESCRIPTION

The system contains a collection of tasks that each implement a single processor. Each processor performs synchronization activity after a certain period called the Synchronization Period. At this point the processor expects to receive clock values from its adjacent connected processors. If the processor does not receive the clock value from one or more of its adjacent connected processors it goes into a state called 'timeout', where the processor waits for a certain amount of time to receive this clock value. Eventually the value either arrives or the wait time expires. In either case, the process eventually proceeds into the synchronization computation state. On completing the computation state, each processor updates its clock to the new clock value using the computed correction value. The mode of communication of the clock values may be either by broadcast or direct transmission to specific adjacent connected processors.

From the above description two types of events are evident, the synchronization event and the timeout event. These events are discrete events occurring at specific times. The nature of occurrence of these events and their ordering based on time makes it possible to implement a simulation system for these events using standard discrete events simulation techniques. Details on discrete event simulation techniques and program components can be obtained from reference[8].

There are certain other tasks that are essential during the startup of the simulation and completion of the simulation. The assumption that all processors are initially assumed to be synchronized within certain limits means that the simulation must start with (relatively) synchronized processors. Hence there is a startup task which ensures the initial synchronization of the processors. A termination task is also required to ensure proper termination of all the tasks on completion of the simulation run.

INPUT AND OUTPUT DATA

Apart from the tasks, the next most important aspects of the simulation system are the inputs it requires and the output it generates. The input data for the simulation defines algorithm-specific data constants, the network configuration, processor attributes, and the number of synchronization cycles the simulation has to run. Among the processor attributes is the specification of "drift" for that processor.

The simulation output contains all data related to system synchronization. This includes, for each processor, the clock differences observed during each synchronization cycle and the correction value computed for each synchronization cycle. The set of correction values provides a measure of drift in the clock over certain period, as well as a measure of the degree of synchronization achieved by a particular algorithm.

PROCESS DESCRIPTION

A set of processors is simulated by an Ada task type. This type is instantiated "N" times to produce "N" (simulated) processors. Each processor contains functions for clock synchronization and communication. The simulation system is implemented on traditional discrete event simulation principles. Each processor is scheduled into synchronization by a scheduler task. The scheduler task picks the top most request from an event queue and schedules that particular processor. An event queue is built upon request from each processor requesting service. The request contains the time at which the service is required and the type of service required by the processor. The event queue is dynamically built by placing the event requests in order of time, contained in the request, from each processor.

Thus the event queue server functions as the driver for the entire simulation.

EVENT QUEUE

The event queue is serviced by the queue_server task or the scheduler. The queue_server picks top most item from the queue and makes a call to the particular processor task activating the function specified in the event item.
In case there is no event posted in the event queue, the queue_server continues to loop waiting for new events to be posted into the event queue.

STARTUP TASK

The startup task has the function of building the initial event queue. The startup task creates an initial event queue with one event for each processor. The initial function for each event is the event 'synchronize'.
The time for synchronization for each processor task is computed based on the synchronization period and drift specified for each processor. This is to satisfy the initial assumption made in the algorithm that initially all the processors are synchronized within certain limits.

TERMINATOR TASK

The terminator task accepts termination signals from the processor tasks. On receiving termination signals from all the processors it terminates the queue_server task. This is done to terminate all the tasks in an orderly fashion, and assure that all tasks are terminated.

PROCESSOR TASK

The processor task contains the processes for synchronization. The call from queue_server instantiates the synchronization process. The synchronization process reads the clock values from the adjacent processes and initiates the error computation function. In case any one of the processor is unable to deliver its clock value the processor initiates a timeout call and posts an timeout event into the event queue. The queue_server reads the timeout event and initiates the clock read and the error computation function. Upon computing the error correction for the clock the clock is updated to the new correct value. It then computes the time for the next synchronization and posts a new synchronization event into the event queue. This continues until the number of synchronizations performed reaches the limit set for the simulation run. Once this limit is reached the processor task initiates a call to terminate, by calling the termination task and registering termination.

DATA STRUCTURES

The simulation system implements "broadcast" communication mode using an array data structure, of size equal to number of processors. This array contains the clock values broadcast by each processor. Another array of the same size is implemented to validate the clock value. Every instant a processors clock value is broadcast, the respective synchronization count is written to validate the clock value. Every time any processor reads the clock values from the broadcast array and finds one of the clock value is not valid it initiates a timeout event. Other data structures are local to respective tasks and functions.

OUTPUT FILE

Each processor creates an output file. The output file contains relevant data to that processor, like the processor id in the network, its adjacent connected node ids, and clock drift for the local clock, and data pertaining to the synchronization computation. Data relevant to synchronization are the clock difference computed between the local clock and its adjacent processor clocks, and error computed by the algorithm as the error correction value for the clock. Data such as clock difference and correction values for the synchronization cycles are recorded into the respective processor's output file.

SYNCHRONIZATION

The synchronization function is implemented as a function in the processor task. This function implements a particular algorithm. Several simulation runs are conducted for different configurations, different drift values, and for all the three algorithms. Data generated is used to study the synchronization pattern with each algorithm, on different processors, with differing configuration and drift rates.

Since the drift value in practice is impredictable, some random number is set which is within a certain range. A normal drift allowed with the clocks in real life is about five to eight cycles, higher or lower for one MHz clock cycle. Hence a drift value within this range is normally selected. Anything beyond this value is considered an error in the clock. Since we can control drift as a simulation parameter any clock can be set with a larger drift to study effect on the synchronization algorithm.

All the three algorithms in this study compute correction values applied to local clocks. These values can be used to compare the quality of algorithm performance, in terms of the fastest convergence of clocks along with the required minimum degree of connectivity for this convergence. Based on the requirement of the algorithms several suitable configurations have been selected for running the simulation. The algorithms are also tested for the number of faulty processors that are tolerated. This is done by introducing drift in more than one processor. In case of more than one faulty processors the degree of adjacency required increases considerably, particularly the LW algorithm, which has implications on the configurations selected for study.

The actual functioning of the simulation system is as follows. Initially the startup task contains the task of building an event queue for all the processors. The first event created for all processors is the synchronization event. The synchronization timing for each processor is set upon considering the drift associated with each processor. The schedular task is activated by this time, which starts reading the event queue and activates appropriate events in the respective processor. Each processor upon completion of the current activated event, posts its next event into the event queue, along with the time value at which this event is to be activated. The queue server continues to pick the next (in time) event from the queue, and activates that event in the appropriate processor. This process continues until a termination condition is satisfied at each of the processor. At this point the termination task is activated which sets the final termination condition for each processor task, terminates the other tasks, then terminates itself, thus completing the simulation run.

The simulation generates data for effective comparison of the algorithms. The primary data of interest are the correction values applied to the clocks. Analysis of this data shows how soon the error value has stabilized close to zero, and in how many cycles this occurs. This gives a very good measure of performance quality for each algorithm. Another data of particular interest is the time taken to execute the algorithms. This is generated by the time taken by the simulation to execute a specific number synchronization cycles. Comparison of the times for different algorithms gives a good measure of overall system overhead.

CHAPTER 5

RESULTS AND CONCLUSION

Next we compare the data generated by our collection of simulation runs. Comparisons are made for drift error correction values generated for the three algorithms, and also for drift observed by processors adjacent to erroneous processors. Several configurations and drift conditions are simulated for comparison. Configurations selected are the hypercube, the array configuration, and a cube of twenty seven processors with 3 X 3 in each plane, with three such planes. Figures 5.1, 5.2, and 5.3 below show the configurations.

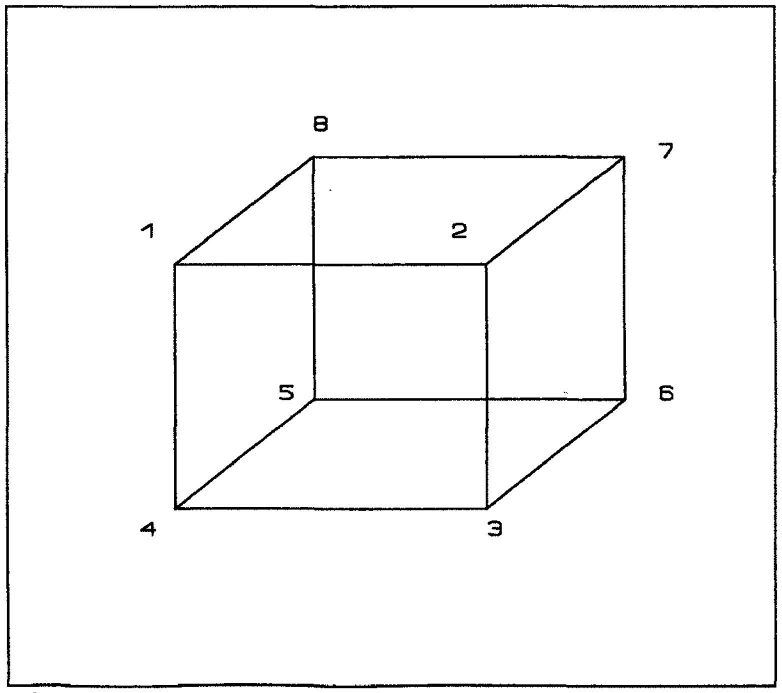

Figure 5.1: HYPERCUBE CONFIGURATION

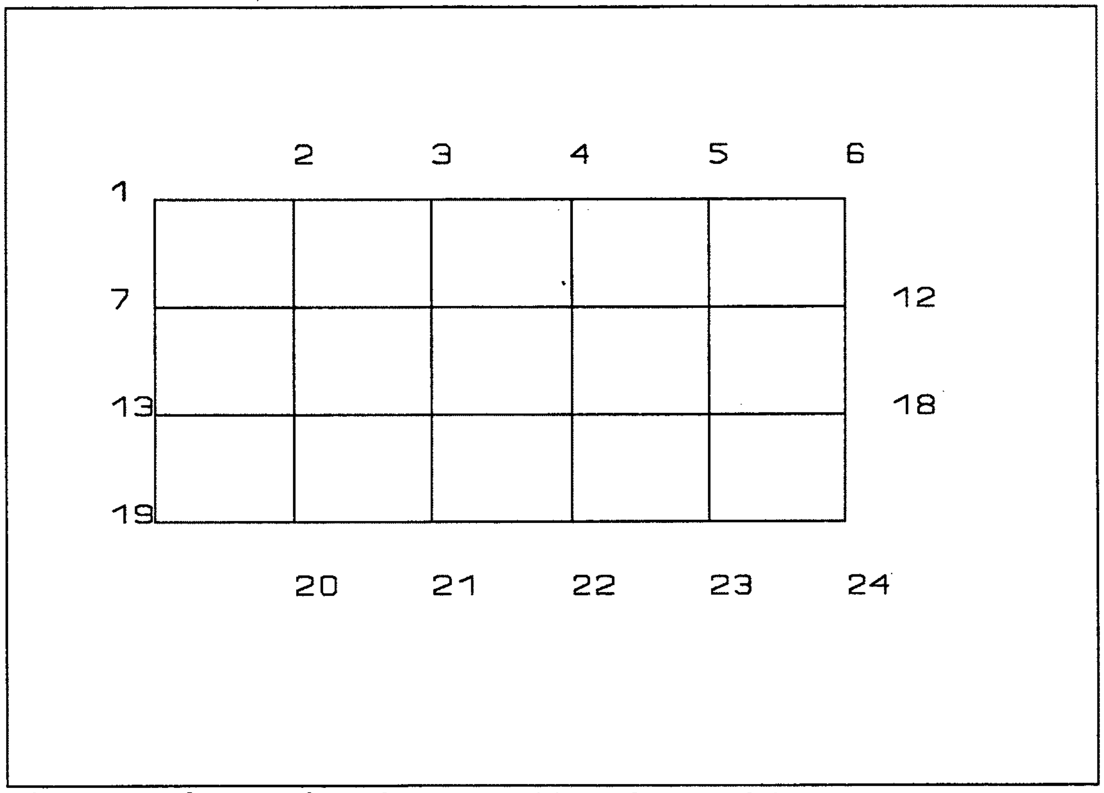

Figure 5.2 : A 6 X 4 ARRAY CONFIGURATION

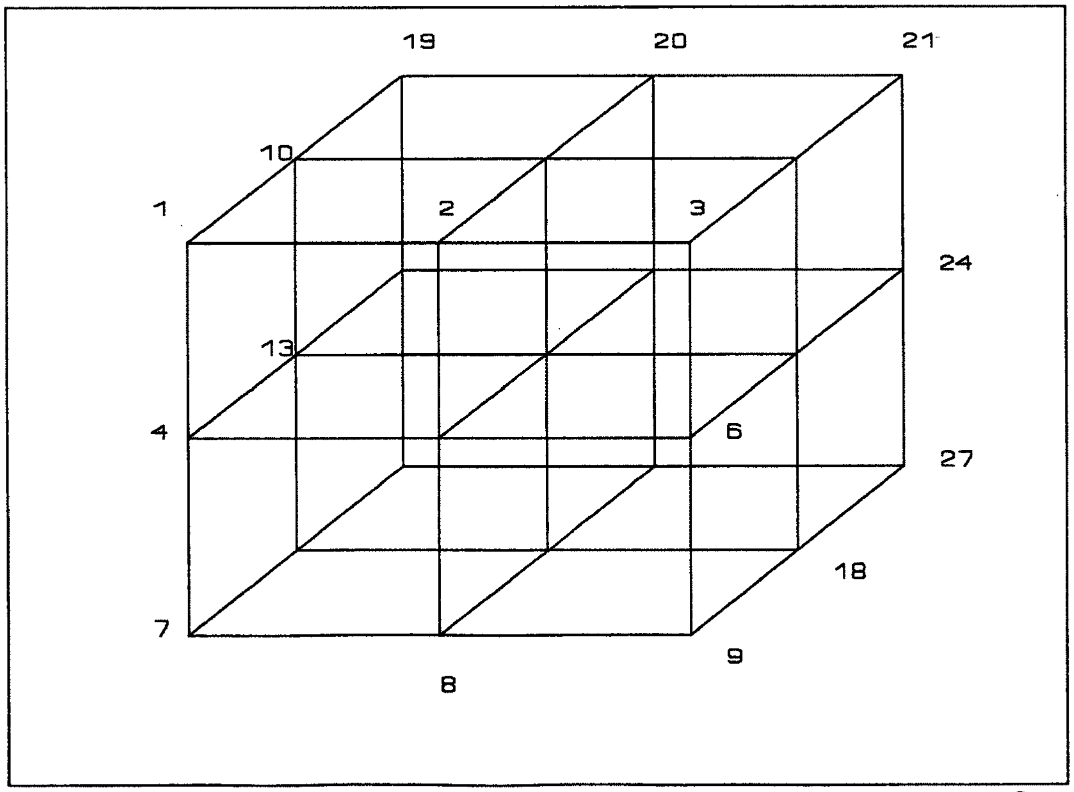

Figure 5.3 : A NETWORK OF 27 PROCESSORS ARRANGED IN 3 X 3 X 3 SETUP

INTERPRETATION OF RESULTS

As the case of standard behaviour of any system, transient behaviour in the system could be expected as the error correction values is computed and applied. Initially as the correction values are applied the processor clock takes few cycles to attain stability for synchronization. The behaviour of processors under both cases could be considered for comparison of the behaviour of the algorithms. As an effect of the transient behaviour large variations in the computed correction values may be expected. This transient nature should not continue for larger number of synchronization cycles. A good measure of the behaviour of an algorithm is how fast the algorithm stabilizes. Under stabilized conditions the correction values computed may continue at a constant level or vary around a central value, with the variations lying within a prespecified limits. The smaller the computed correction value, or the smaller the variation of the computed correction value, the better the performance of the algorithm. Based on these factors we continue to compare the results of the simulation run plotting graph for the error correction value computed in Y axis, verses the synchronization cycles in X axis.

Graphs depicting error correction value versus number of synchronizations are shown for selected processor nodes. In general the graphs describe drift and correction at a specific node in a particular configuration. Performance is compared for each of the three algorithms under the same conditions.

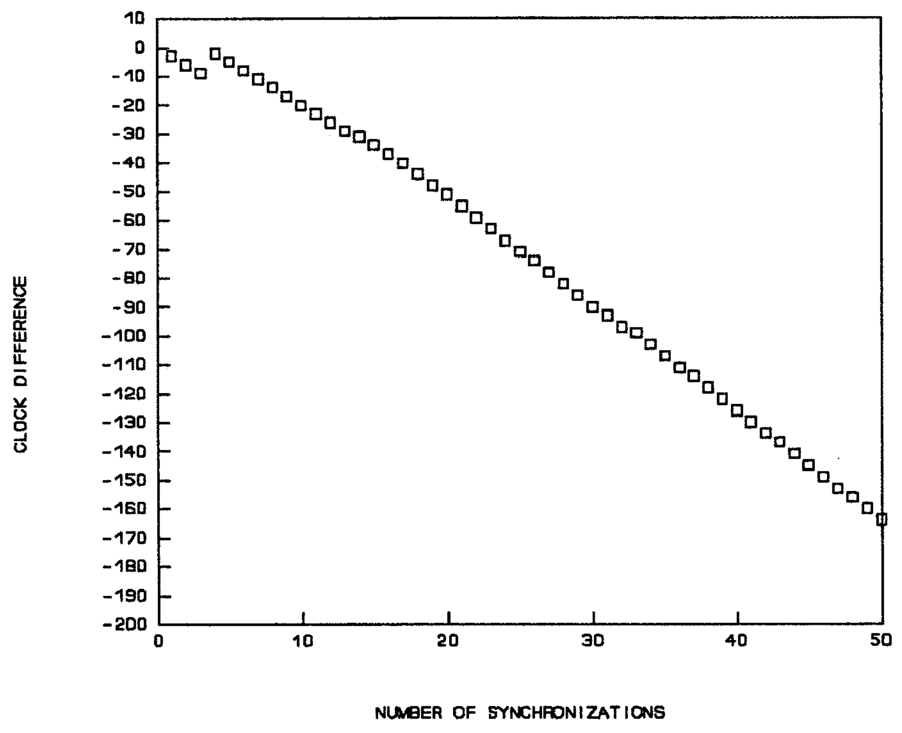

Figure 5.4 : Drift without any correction applied

Figure 5.4 shows how a clock with drift moves away from real time with passage of time if no correction is applied. The clock difference increases with the number of synchronizations. In this specific case the drift in the processor is negative hence the clock values is lower than the real time.

Figure 5.5 compares the performance of the three algorithms, for reference node 4 in the Hypercube configuration with a drift of -0.000003. If the values are within acceptable limits the performance of the algorithm is acceptable, but a better performance is represented by smaller error correction values and faster stabilization of the clock. As shown in this graph, algorithm CNV attains stability faster than algorithm GH, which stabilizes faster then algorithm LW. The error correction values computed by algorithms GH and LW on attaining stability are the same. Hence we conclude that in terms of attaining stability algorithm CNV is better than GH, which is better than algorithm LW. The graph shown here is representative of the results obtained for this configuration and various values of drift upto the limit of ±0.000005. This scenario represents a normal condition with tolerable drift limit.

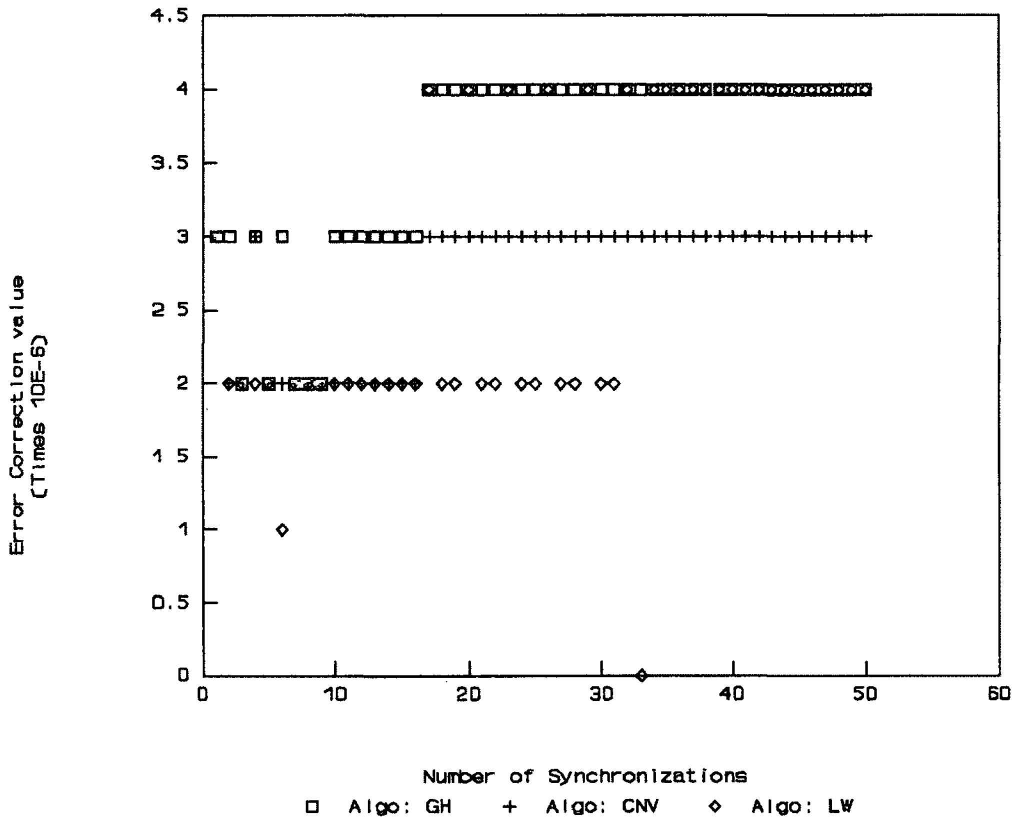

Figure 5.5 : A sample graph showing drift correction values over synchronization cycles.

Figure 5.6 is a case of processor with unacceptable drift. The graph shows the clock difference perceived by processor 3 and the drifty processor 4. Processor 3 is set as a healthy processor without any drift and processor 4 is set with a drift of -0.000009. The graph represents clock difference between processor 3 and 4 for the three algorithms as computed in processor 3. In case of algorithm CNV, processor 4 because of its large drift rejects clock values received from its neighbors due to a large difference between the clocks. This forces the processor to continue with the same clock without applying any correction to the clock. This result justifies the assumption made about this algorithm that if the local clock is faulty the processor fails to synchronize. Thus with algorithm CNV we see that the clock difference increases without any control. In contrast, with algorithm GH and LW since the clock filtering is not done based on the local clock the process corrects the clock values and hence we see a smaller clock difference. The configuration for this graph is an hypercube.

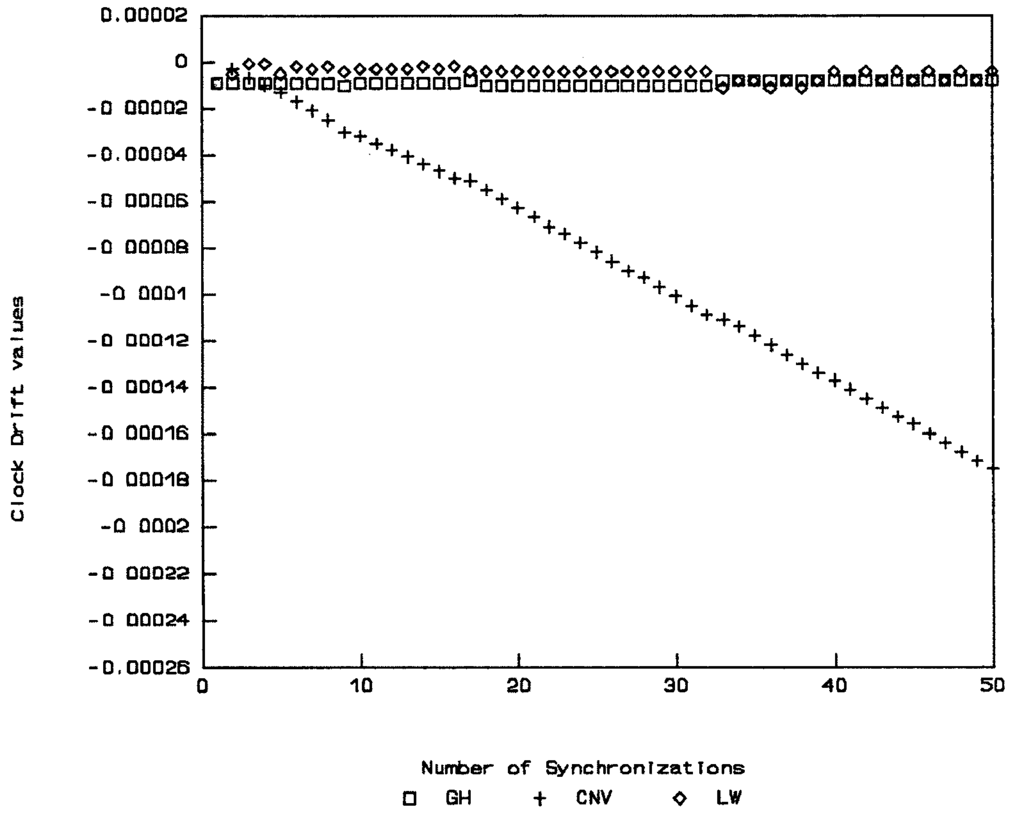

Figure 5.6 : Graph showing drift observed by processor 3 in terms of clock difference between processor 3 and 4. Drift in 4 = -0.000009.

Figures 5.7, 5.8 and 5.9 are graphs for the three algorithms GH, CNV and LW respectively. The specific condition is a situation with two processors drifting in a hypercube configuration. One processor is set with a larger drift, and the second almost to the limit. The drift is as observed from a processor which is adjacent to both of the drifting processors. The situation is viewed from processor 5, a processor adjacent to the drifty processors. The graphs represent the clock differences between processor 4 and 5 and processor 6 and 5, and the error correction value computed at processor 5. From the three graphs we see that only algorithm GH continues to keep processor 5 synchronized despite its adjacent processors being erroneous. The graphs show the requirement of minimum connectivity required for the processors to synchronize. Figure 5.9 shows the case for algorithm LW that processor 5 fails to synchronize, forced due to its two faulty neighbors. A similar situation in case of GH has processor 5 still synchronized.

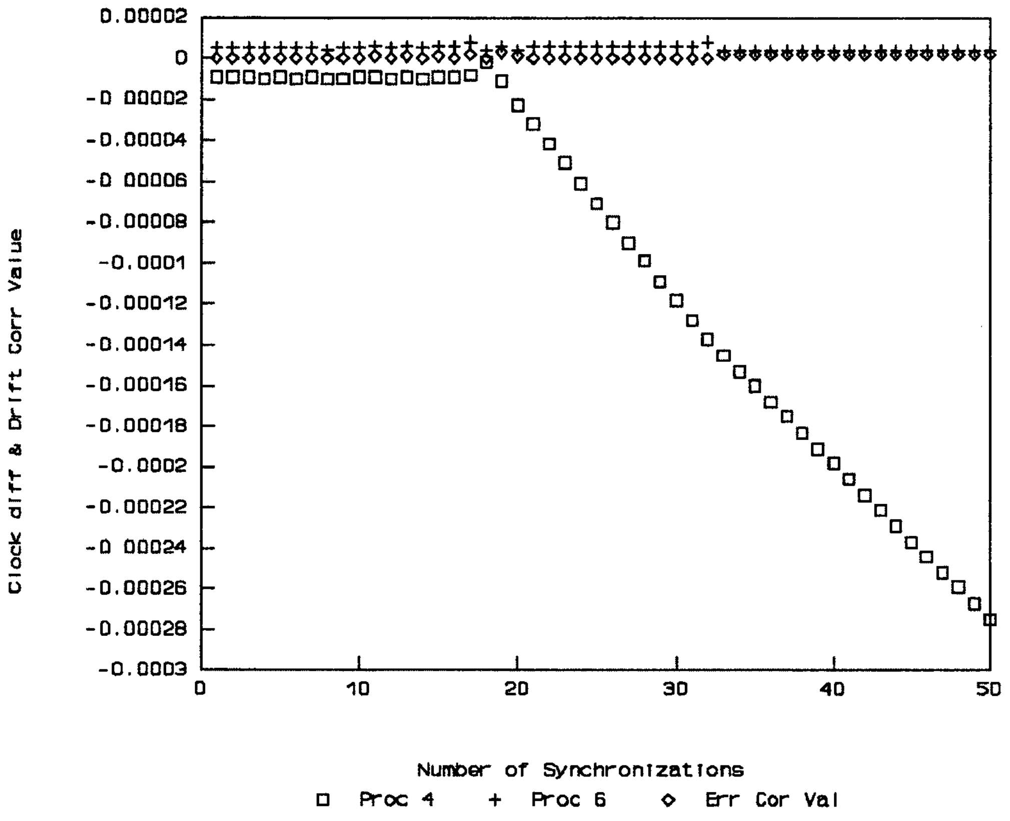

Figure 5.7 : Graph showing clock difference values and drift correction for a specific processor. Ref prof: 5, proc 4 drift = -0.000009, proc 6 drift = 0.000005. Algorithm GH

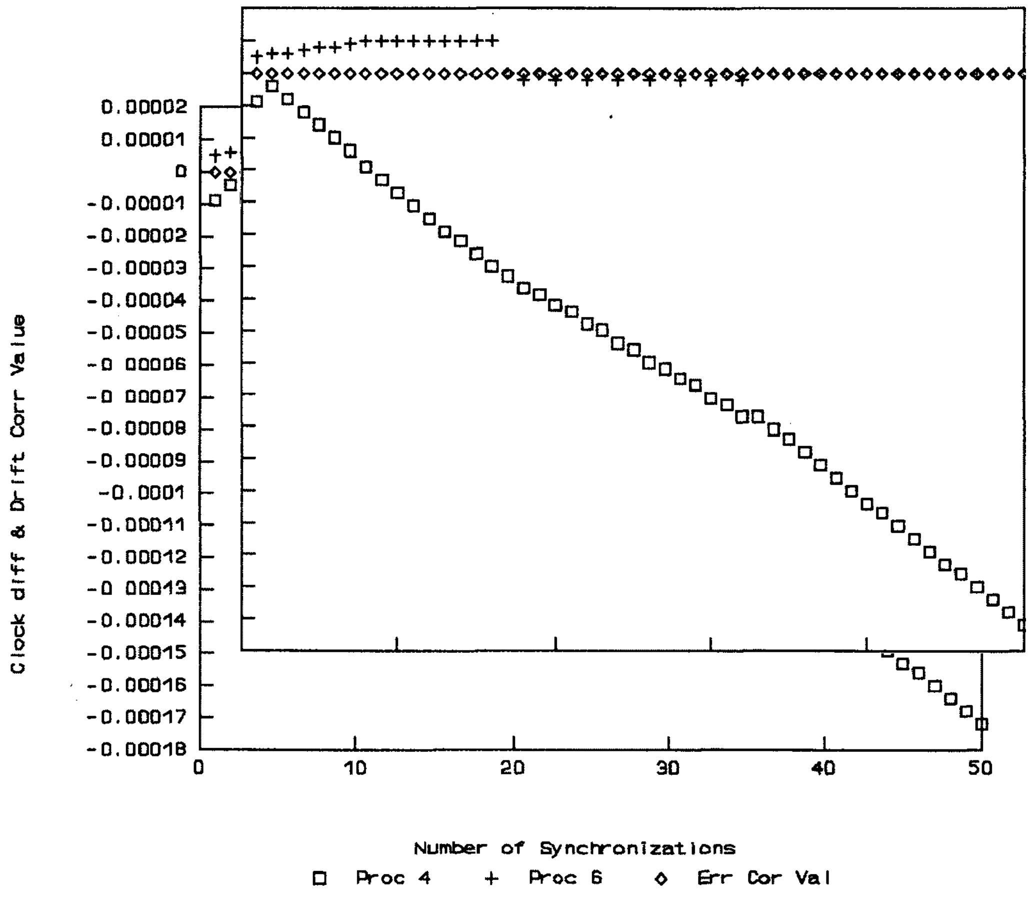

Figure 5.8 : Graph showing clock difference and error correction value in processor 5. Proc 4 drift = -0.000009, Proc 6 drift = 0.000005, Algorithm : CNV

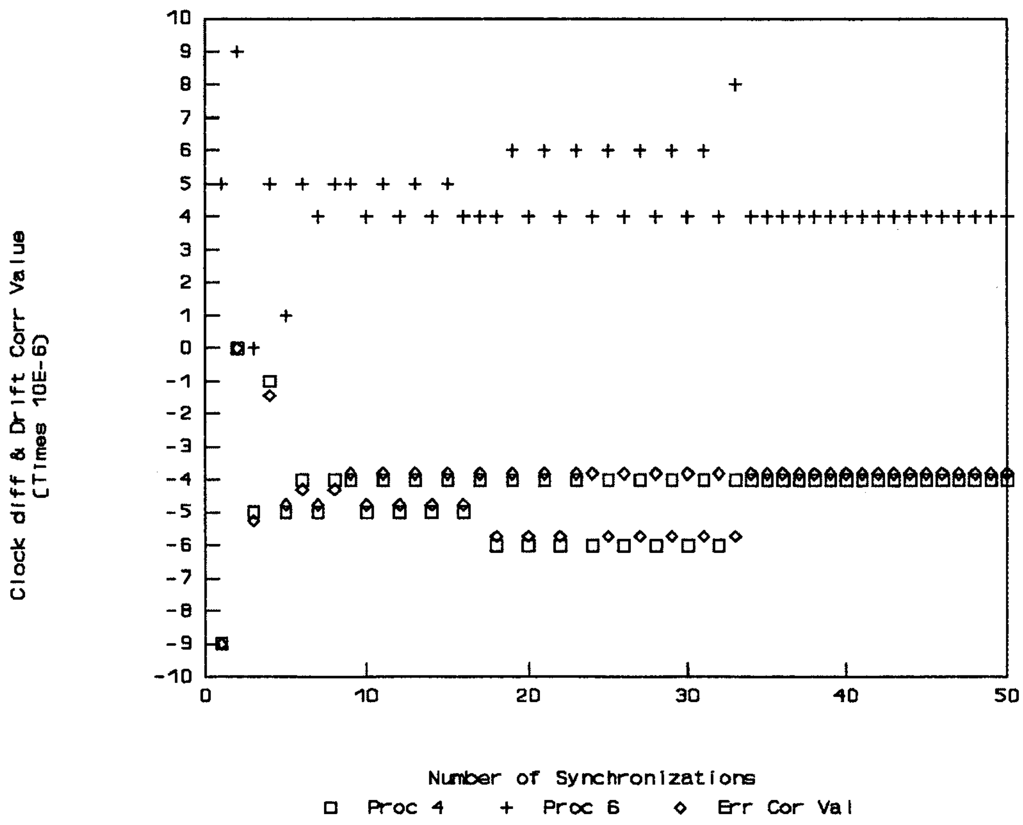

Figure 5.9 : Graph showing clock difference and drift computed at processor 5. Proc 4 drift = -0.000009, Proc 6 drift = 0.000005. Algorithm : LW

The next three graphs, Figures 5.10, 5.11, and 5.12, show a sample of range of the error correction values computed by the processors. The graphs are for the algorithms GH, CNV and LW respectively. The configuration is the hypercube configuration. The graphs compare the error correction values and a sample range of these values as computed by different processors.

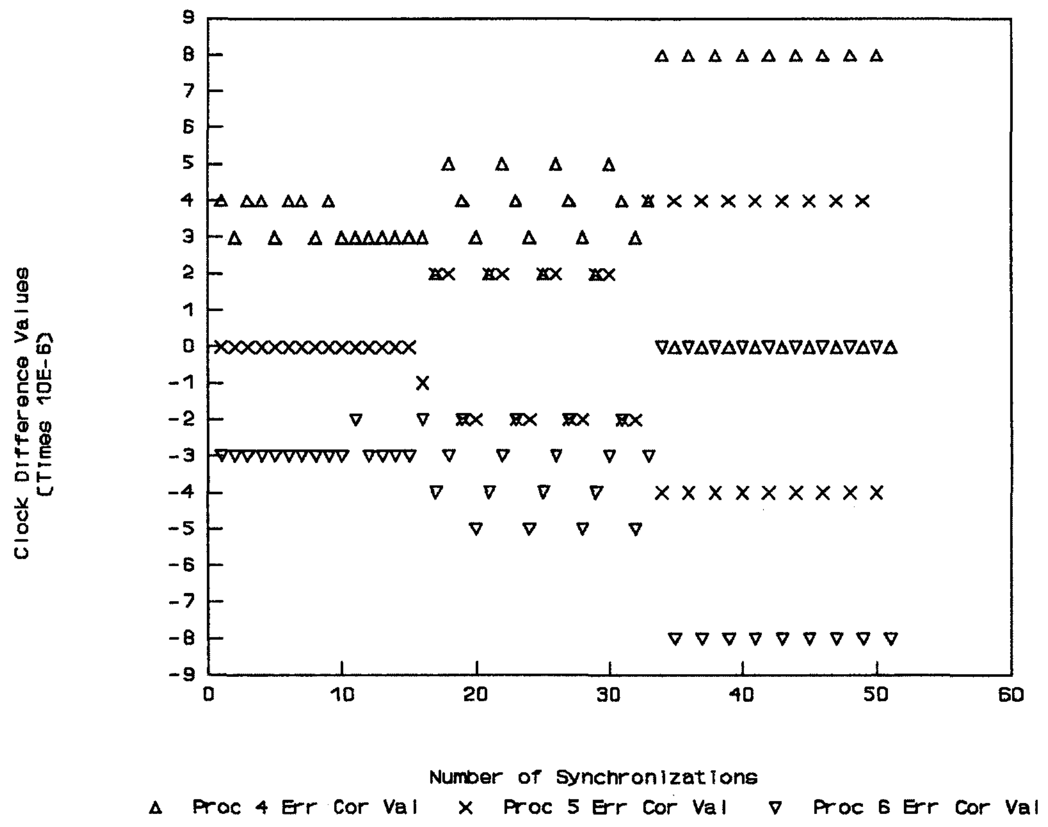

Figure 5.10 : Range of error correction values computed by the processors. Processors : 4, 5 and 6. For Algorithm GH

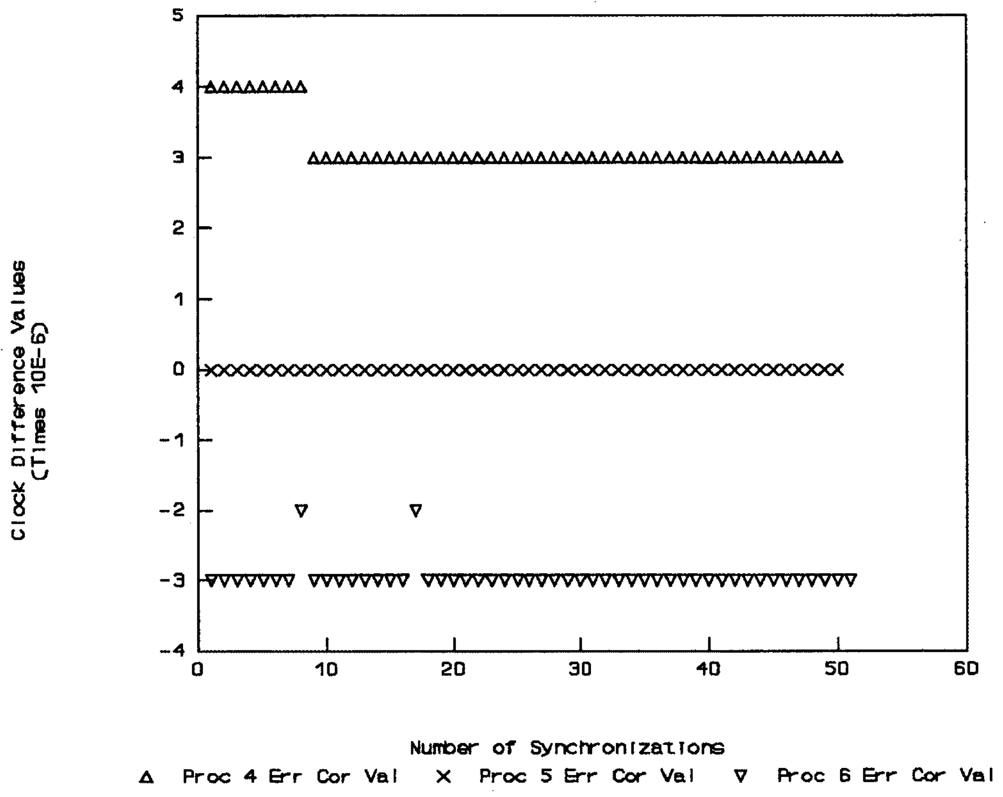

Figure 5.11 : Range of error correction values computed by different processors. Processors: 4, 5 and 6. Algorithm : CNV

In case of Figure 5.10 the transient behaviour though lies well within safe limits the stabilized condition shows the correction value about to fail. The correction values for processor 4 swings between 0 and +8 cycles and for processor 4 swings between 0 and -8. This is a case of processor operating within its limits. Under similar circumstances for algorithm CNV and LW the behaviour is vary well within limits and algorithm CNV and LW's performance is better in this specific case.

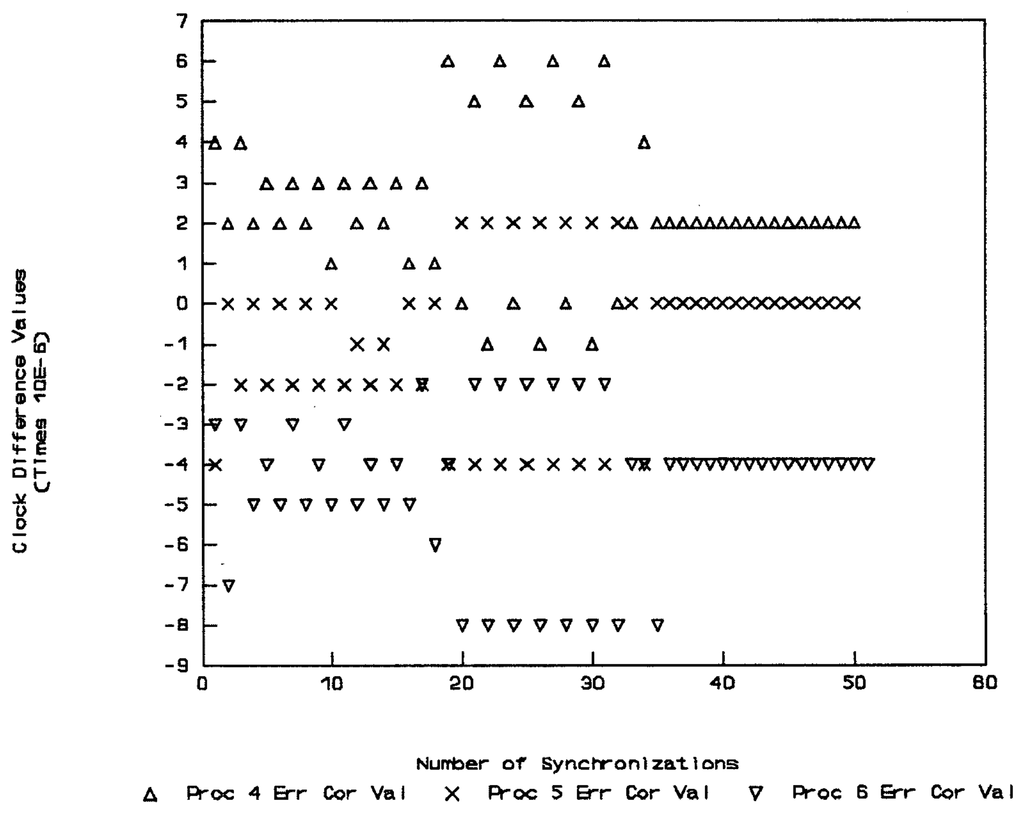

Figure 5.12 : Range of error correction values computed by processors. Processors: 4, 5 and 6. Algorithm : LW.

Figures 5.13 and 5.14 are for an array network of 24 processors arranged as a 6 X 4 network. The graphs show a comparative performance of the three algorithms in node 10 and 15 of the network. The graphs show the performance of the algorithms. Algorithm GH has a larger value of error correction values but is still within the acceptable range. The overall performance is comparable to the other two algorithms.

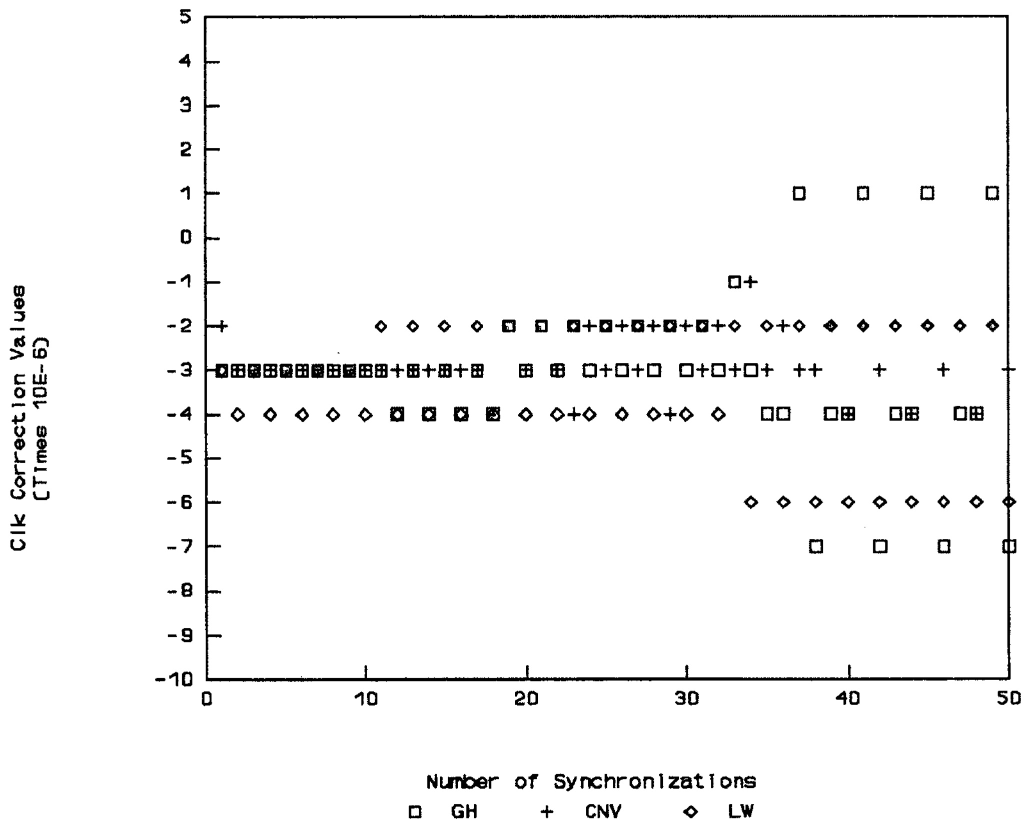

Figure 5.13 : Graph comparing error correction value computed by the three algorithms in a 24 processor 6 X 4 network. Reference processor : 10 , drift : 0.000003

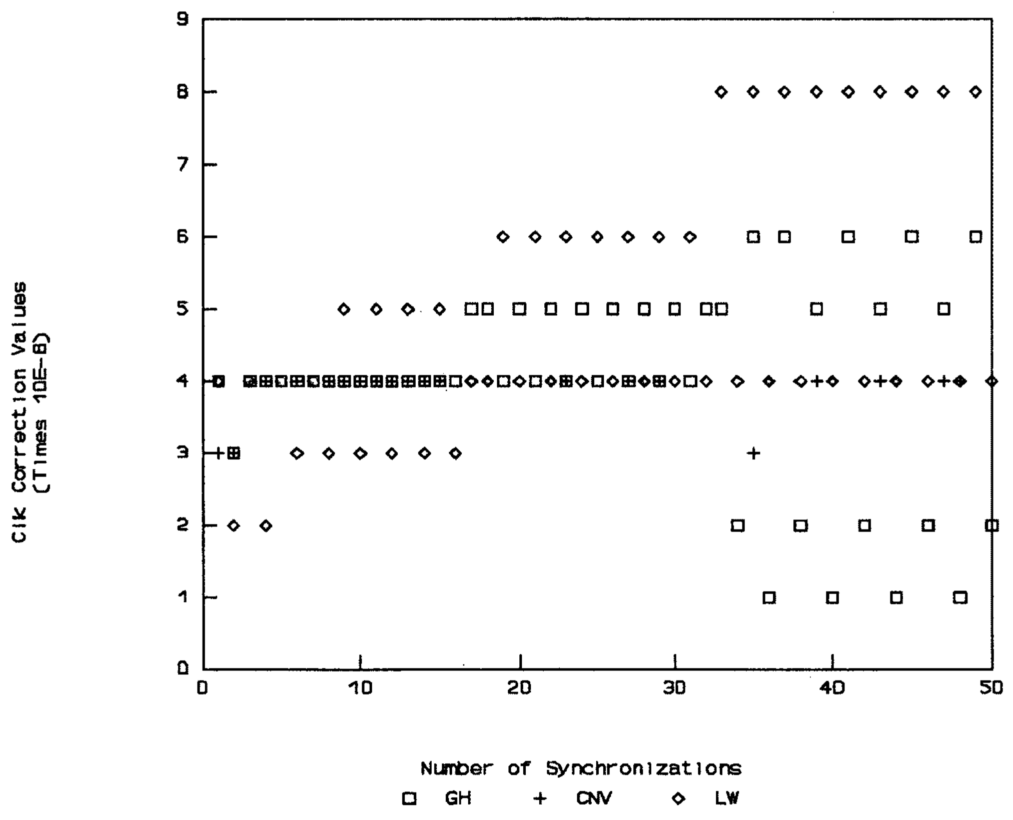

Figure 5.14 : Graph comparing error correction values computed by the three algorithms at node 15 with drift of -0.000004 in a 24 node 6 X 4 network.

Figure 5.15 shows a sample input file for the simulation run. The file contains the number of processors in the configuration, and the number of synchronizations the simulation executes. Other information like the adjacency information among the processors is input as "adj_matrix" and the drift specification for each processor is input through the "drift_spec_array". The "mul_matrix" array contains the number of processors connected to each processor. The index for an element of the array acts as an id for the processor to which this value is associated.

```
SYNCHCOUNT : integer := 50;  -- number of synchronization cycles to
set termination
NUM_OF_PROCESSORS : integer := 8; -- ********* NUM OF NODES IN SETUP

EXPT_NUM : integer := 6;



adj_matrix : array ( 1..num_of_processors , 1..num_of_processors ) of
integer:=
     (  ( 6, 4, 2, 0, 0, 0, 0, 0 ),
        ( 1, 3, 7, 0, 0, 0, 0, 0 ),
        ( 2, 8, 4, 0, 0, 0, 0, 0 ),
        ( 1, 3, 5, 0, 0, 0, 0, 0 ),
        ( 4, 6, 8, 0, 0, 0, 0, 0 ),
        ( 5, 7, 1, 0, 0, 0, 0, 0 ),
        ( 2, 6, 8, 0, 0, 0, 0, 0 ),
        ( 5, 3, 7, 0, 0, 0, 0, 0 ) );


drift_spec_array : array(1..num_of_processors) of real :=
                    ( 0.000000,  0.000000,  0.000000, -0.000004,
                      0.000000,  0.000003,  0.000000,  0.000000 );


mul_matrix : array ( 1..num_of_processors ) of integer :=
     ( 3, 3, 3, 3, 3, 3, 3, 3 ); -- ******* MULTIPLICITY ARRAY
```

Figure 5.15: Sample input data file showing configuration data.

Conclusions:

The data collected from the simulation is used to verify the initial arguments made about the proposed algorithm. Arguments made about the algorithm GH as being

1. More or equally robust in terms of fault tolerance than LW and CNV
2. Less overhead in terms of requirement of minimal connectivity for satisfactory synchronization,
3. Tolerance against faults in local clocks, has been proved.

This is rightly indicated by the graphs in Figure 5.6, 5.7, 5.8 and 5.9. The failure of processor with fault in local clock to synchronize, with algorithm CNV for synchronization is shown in Figure 5.6. The graphs from Figure 5.7 proves that GH is tolerant against local clock failure thus overcomes a major drawback from algorithm CNV. Figure 5.8 and 5.9 clearly shows how GH over comes the requirement of minimal connectivity despite being connected to two faulty processors and still continuing to synchronize, where under similar situation algorithm LW fails to do so. The discussion from comparing the graphs clearly show comparable to better performance of algorithm GH over the other two algorithms in worst case situations, as well under normal situations. Hence algorithm GH proves its merits over the other two algorithms.

REFERENCES

1. Leslie Lamport, "TIME CLOCKS AND THE ORDERING OF EVENTS IN A DISTRIBUTED SYSTEMS", Communication of the ACM, July 1978.

2. Parameshwaran Ramanathan, Kang G. Shin, Ricky W.Butler, "FAULT TOLERANT CLOCK SYNCHRONIZATION IN DISTRIBUTED SYSTEMS", Computer 1990 October.

3. Leslie Lamport, P.M.Mellier Smith, " BYZANTINE CLOCK SYNCHRONIZATION ", ACM, June 1984.

4. Anne Dinning, " A SURVEY OF SYNCHRONIZATION METHODS FOR PARALLEL COMPUTERS", Computer, July 1989.

5. " REAL TIME SYSTEM DESIGN ", Levi, Agrawala, McGraw Hill Pub.

6. Jennifer Lundelius Welch and Nancy Lynch, " A FAULT - TOLERANT ALGORITHM FOR CLOCK SYNCHRONIZATION ", Information and Computation Vol 77, Number 1, April 1988.

7. Leslie Lamport and P. M. Melliar-Smith, " SYNCHRONIZING CLOCKS IN THE PRESENCE OF FAULTS ", Journal of the Association for Computing Machinery, Vol 32, No. 1, January 1985.

8. " Discrete Event Systems, Modeling and Performance Analysis ", Christos G. Cassandras, Aksen Associates Inc., Publications.