Trans. Nat. Acad. Sci. & Tech. (Phils.)
1987.9:51-63

DISTRIBUTED CONTROL USING A HIERARCHY OF COORDINATORS

Arturo I. Concepcion
College of Computer Studies
De La Salle University
Taft Avenue, Manila. Philippines

ABSTRACT

One of the central issues in the efficient and reliable operation of distributed computer systems (DCS) is distributed control. This issue deals with the decentralization of the control of the entities that provide the many functions of DCS. The absence of a central coordinator forces the entities to rely on timely information provided by other entities to decide on the best choice of action. But this scheme cannot be solved in real-time because of the overhead in computation and communication. Furthermore, the presence of unreliable communication and delays add more complexity to the issue. This paper proposes the use of a hierarchy of coordinators to solve the distributed control issue of DCS. The function of the hierarchy of coordinators is to ensure the timely arrival of information and to provide the global state of the DCS to the entities. The construct of coordinators also guarantees the computation of the decision process in finite time. This paper presents the distributed control algorithm based on the construct. To study its behavior, a test bed was implemented on a multiprocessor architecture, the Heterogeneous Element Processor (HEP). Simulation runs made on the HEP showed that the construct is viable solution.

Introduction

A current issue that generates much interest in research is the distributed control in distributed computer system (DCS). DCS offers several advantages over Von Neumann type of computers. Among the advantages are computation speed-up, high reliability, resource sharing, better utilization and extensibility [STAN84]. To achieve high reliability in DCS, functions such as scheduling algorithms, deadlock detection, routing and congestion control algorithms and concurrency control algorithms are distributed over several decision makers on nodes in the DCS. But having no central coordinator, control over these functions becomes a difficult task. Having all of these algorithms coordinate to achieve good overall performance is even more complex. Ho in 1980 [HO80] proposed to solve this problem through team decision scheme. But this algorithm involves a large overhead in terms of computation and communication. To reduce the overhead, estimation of the global

state requires some relaxation to the team decision scheme such as step delay approaches [HO80] and periodic coordination [LARS82].

This paper proposes a structure, the hierarchy of coordinators, to achieve distributed control in a DCS. This structure will guarantee that the computation of the global state can be done in finite time and satisfy time constraints, in the case of real-time systems. An overview of some of the proposed algorithms to maintain the correct and efficient operation of the DCS is presented in section 2. Section 3 of this paper discusses the mechanism of the hierarchy of coordinators in maintaining distributed control over a set of processes. Then section 4 presents the implementation of a test bed for the hierarchy of coordinators on the Heterogenous Element Processor (HEP). This machine is an MIMD computer. Simulation runs and analysis are described in section 5. Finally, Section 6 offers some future directions on our approach.

Deadlock, Global State and Roll-Back Issues

In a distributed environment, there are issues that must be addressed such as deadlock detection, global state determination and roll-back mechanism. These issues constitute a substantial overhead on the correct and efficient operation of the DCS. Research efforts have been directed to answer these issues in the past, [JEFF85, CHAN83, CHAN85, OBER82, MENA79].

For deadlock detection in a DCS, algorithms were proposed in [CHAN83]. The algorithms consist of an idle process initiating a deadlock query message to its dependent set. The dependent set consists of a linear order of processes where P_i is idle and is waiting for a resource held by Pi+1. Process P_j is dependent on process P_k if there exist a dependent set between processes P_j and P_k. P_j is deadlocked if it is dependent on itself. For every deadlock detection computation, the maximum number of messages sent for N processes is N*(N-1). Other distributed deadlock algorithms involve the construction of the hierarchical wait-for graph [MENA79, OBER82]. The global wait-for graph is constructed from lower level coordinators which form the wait-for graph of their respective constituents. This information is gathered by higher level coordinators until a single coordinator (the root) computes the wait-for-graph for the whole system.

In [CHAN85] an algorithm was proposed to determine the following state in a DCS: a computation has terminated, or the system has deadlocked, or a token has disappeared in the ring. The algorithm to detect any of the above conditions consists of a process (after recording its state) sends a marker along each channel incident to and directed away from the process. Each process receiving the marker on the channel will record its own state and the marker is sent along its incident channel. The algorithm terminates when all the channels have been traversed. To ensure that the computation of the global state is terminated in finite time, no marker will remain forever in a channel and its process will record its state in finite time.

Finally, [JEFF85] presented a roll-back scheme for the system of concurrent processes which does not synchronize. Each message has a timestamp and each process has a local virtual time. The process executes these messages according to a timestamp ordering. Since the process does not synchronize its execution with other processes, a straggler message (timestamp is less than local time of process) might arrive at some time later. When this happens, the scheme will initiate a roll-back to undo the previous computation back to a consistent state. This state corresponds to the state of the process just before the time indicated on the timestamp of the straggler message.

The determination of the global state of the system of processes is a hard task to perform since the processes are at a different local clocks and the processes communicate at will. The algorithm presented by [CHAN85] determines only specific types of global states (termination of computation, deadlock and loss of token) the DCS might be in. Deadlock can occur if there are no constraints in the manner in which the processes can wait for another process holding a resource. Allowing also a process to go ahead of its computation regardless of whether the process is in synchrony with other processes or not, results in the necessity of a roll-back scheme for re-synchronization. The algorithms presented in [CHAN83, CHAN85, JEFF85] contain overhead which could lead to unacceptable performance depending on the frequency of its invocation and the extent of rolling back.

Hierarchy of Coordinators

This paper proposes a distributed control algorithm for a DCS. The DCS is modelled as a system consisting of a set of processes communicating by message passing. There are no shared variables and each process is assumed to reside completely in a processor. Thus the terms process and the processor will be used interchangeably in this paper. The communication link between processes is assumed to be reliable and the probability of being down is remote.

Let $P = \{ P_i \}$ be the set of processes,
$i = 1, 2, \ldots, n$ where $| P | = n$.
Let $C = \{ C_{ij} \}$ be the set of coordinators,
$i = 1, \ldots, \gamma = 1, 2, \ldots, \eta$
where γ is the maximum level of hierarchy
and η is the number of partitions,
Let $[P = \{ [P_i \}$ be the set of partitions,
$i = 1, 2, \ldots, \eta$ and
$[P_i \epsilon P$ if level $= 0$
$[P_i \epsilon C$ if level > 0.

This assignment of coordinators to partitions of processes and coordinators produces a hierarchy of coordinators, see Fig. 1. The coordinator assigned to the root is $C_{\gamma\eta}$ where γ is the maximum level of hierarchy and η is the number of partitions.

It was mentioned earlier that a process completely resides in a processor. The coordinators may or may not be assigned to a separate processor. Thus several coordinators might reside in one processor. The hierarchy of coordinators, from level 1 to γ, is a logical interconnection among coordinators. To simplify our discussion, however, the coordinators are assumed to reside in separate processors.

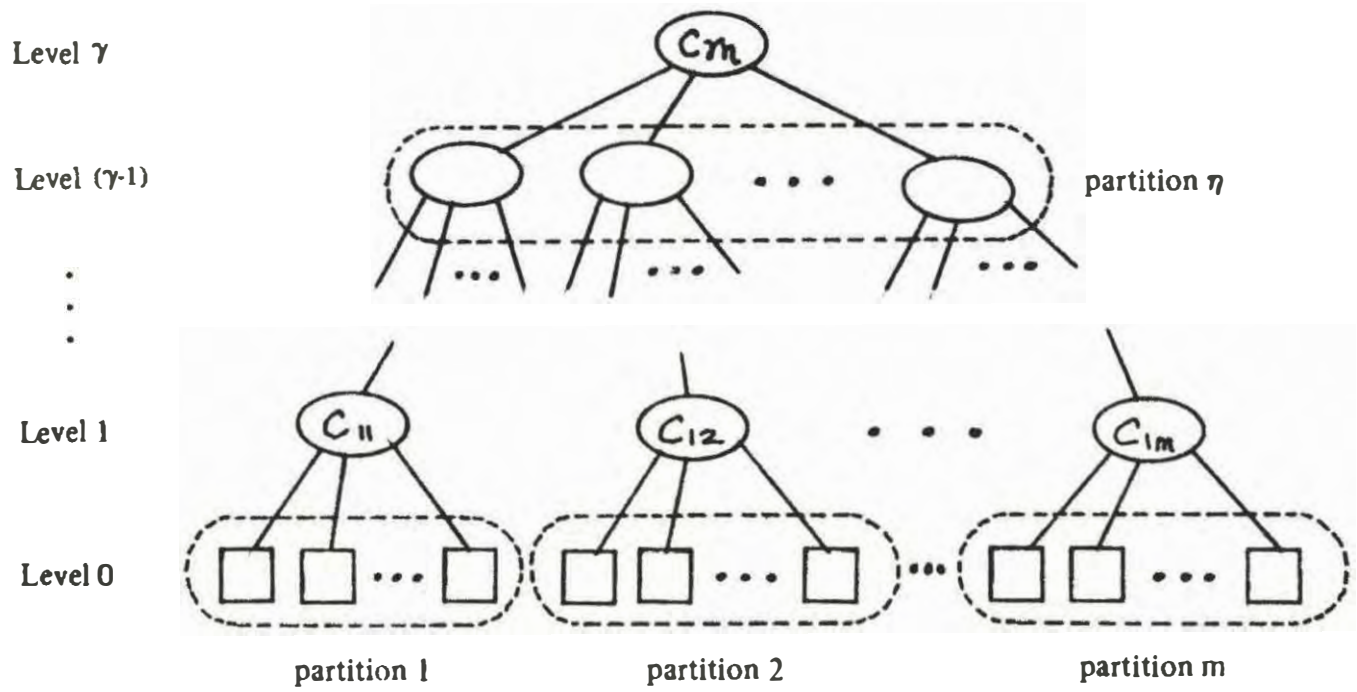

Fig. 1

3.1 Algorithms for a Coordinator and a Process

Each process has the following state variables:

t_L = timestamp of the last message received or time of last signal received.

t_N = estimate of the next time a process will send a message.

s = current state of a process.

A message contains 4 tuples, (spid, dpid, τ, msg), the source and destination process(es), the timestamp of the message and the actual message itself. Note that the timestamp on the message is the current global time. Whenever a process receives a message, the process' t_L is updated to τ and an estimate of the next time the process will send a message, t_N, will be computed. The process with the minimum t_N is allowed to communicate its results to another process(es). The process with the minimum t_N is called the imminent process.

The algorithm for coordinators ensures synchronization among processes by sending appropriate messages to its subordinates and to its upper level coordinator, if any. The coordinators maintain the following state variables:

T_L = timestamp of the latest message received or timestamp of the latest signal received (from an upper level coordinator).

T_N = the minimum t_N of its subordinates (which is either processes or coordinators).

S = collected states of subordinates.

Each coordinator has the following responsibilities: to route and scrutinize messages for consistency; to collect the states of its subordinates and build a partial global picture of the DCS; to synchronize its subordinates; and to guarantee that the global state computation terminates. The algorithms are shown in Figs. 2 and 3. Each algorithm consists of the actions taken when receiving an external message, emsg; a signal from a coordinator; or an output message from a process, ymsg, from subordinates. Note that the algorithms discard messages which are out of synchrony with the system.

The following gives a summary of the actions taken by either a process or a coordinator when it receives a message.

When a process receives a signal, it first checks whether the message time-stamp, τ, is within the range of allowable values. The process computes its output ymsg to its coordinator. Simultaneously, the process computes its t_N

Algorithm for a Coordinator

```
begin
   recv(spid, dpid, τ, msg)                    [ message is received ]

   select
   (T_L < = τ < = T_N and type (msg) = emsg)
   begin
      send (spid, dpid, τ, emsg)               [ send to all processes ]
      for all processes ϵ dpid                 [ in the set dpid and ]
         recv (spid, dpid, t_N, s)             [ wait until all the ]
         store T_N and s                       [ processes send their ]
      end for                                     [ t_N and s ]
   end

   (or τ = T_N and type (msg) = signal )
   begin
      send (spid, dpid, τ, signal)             [ where dpid is the ]
      parbegin                                 [ imminent process ]
         begin
```

```
            recv (spid, dpid, t_N, s)          [ received states from ]
            store t_N and s                    [ processes ]
         end
         begin
            recv (spid, dpid, τ, ymsg)         [ received out - ]
            if dpid ϵ subordinates then        [ put msg ]
            begin
               send (spid, dpid, τ, emsg)      [ send ymsg as ]
               for all processes ϵ dpid        [ emsg to the ]
                  recv (spid, dpid, t_N, s)    [ destination ]
                  store t_N and s              [ processes and ]
               end for                            [ wait for ]
            end                                   [ results ]
            else send (spid, dpid, τ, msg)     [ else, send ]
         end                                   [ msg to next ]
      parend                                   [ level coordi- ]
   end                                         [ nator ]

      (otherwise)                    [ if not within range, disregard ]
      begin                          [ message and continue ]
         discard msg
         return
      end
   end select

   T_L = τ                                    [ compute the state ]
   T_N = min (t_N of subordinates)            [ variables of the ]
   S   = compute state (s of subordinates)    [ coordinator and ]
   send (spid, dpid, T_N,S)                   [ send results ]
end.
```

Fig. 2

Algorithm for a Process

```
begin
   recv (spid, dpid, τ, msg)                    [ message was received ]
   if (t_L <= τ <= t_N) then t_L = τ
      else
         begin
            discard msg                         [ if not within range ]
            return
         end                                    [ ignore the message ]

   select
      (τ <= t_N and type (msg) = emsg)                  [ compute ]
         s = δ_ext (s, emsg)                            [ state ]

      (τ = t_N and type (msg) = signal)
      parbegin
         begin
            y = λ (s)                           [ compute output and ]
            send (spid, dpid, τ, ymsg)          [ send result to ]
         end                                    [ processes ]
         s = δ_int (s)                          [ compute state ]
      parend

   end select

   t_N = t_L + α(s)                             [ compute t_N and ]
   send (spid, dpid, t_N,s)                     [ send result, including s, ]
end.                                            [ to coordinator ]
```

Fig. 3

and its new state, s, which are sent to the coordinator.

When a coordinator receives a signal it checks first the value of τ for synchronization consistency. The signal is sent to the imminent subordinate (either a process or coordinate). After the signal is sent, the coordinator waits for the imminent subordinate to send its new state variables. Then the coordinator determines the new imminent subordinate.

When a process receives a emsg, it first checks τ for synchronization consistency. The coordinator then sends emsg to all destination subordinates. The coordinator then waits for all destination subordinates to send their new state variables. After which the coordinator determines its new state variables and the new imminent component.

When a coordinator receives a ymsg, it determines whether this message is destined for a subordinate(s) or not. If the destination is a subordinate(s), then the coordinator sends ymsg as a emsg, otherwise, the coordinator sends the ymsg to its upper level coordinator.

Implementation of a Test Bed on the HEP Computer

The architecture of the Heterogeneous Element Processor (HEP) has been described in [GAJS85, HWAN84]. As shown in Fig. 4, the main components are the Data Memory Module, the Packet Switch Network and the Process Execution Module. The HEP architecture is classified as a MIMD machine which can execute multiple instructions on multiple streams of data. A program consists of one or more tasks while each task consists of one or more processes. Each process is composed of a sequence of instructions. Both the tasks and processes are executed in parallel in HEP while the instructions of each process are executed in a sequential pipeline fashion. Each PEM has a program memory where active tasks and process instruction streams are selected for execution. Up to 50 instruction streams can be active at any given time. Notice that each PEM has a number of functional units which allow pipeline execution of multiple instruction streams for multiple data streams. For software support, HEP has the DENELCOR's Extended FORTRAN 77 [DENE 84]. This provides the parallel programming environment for the HEP computer.

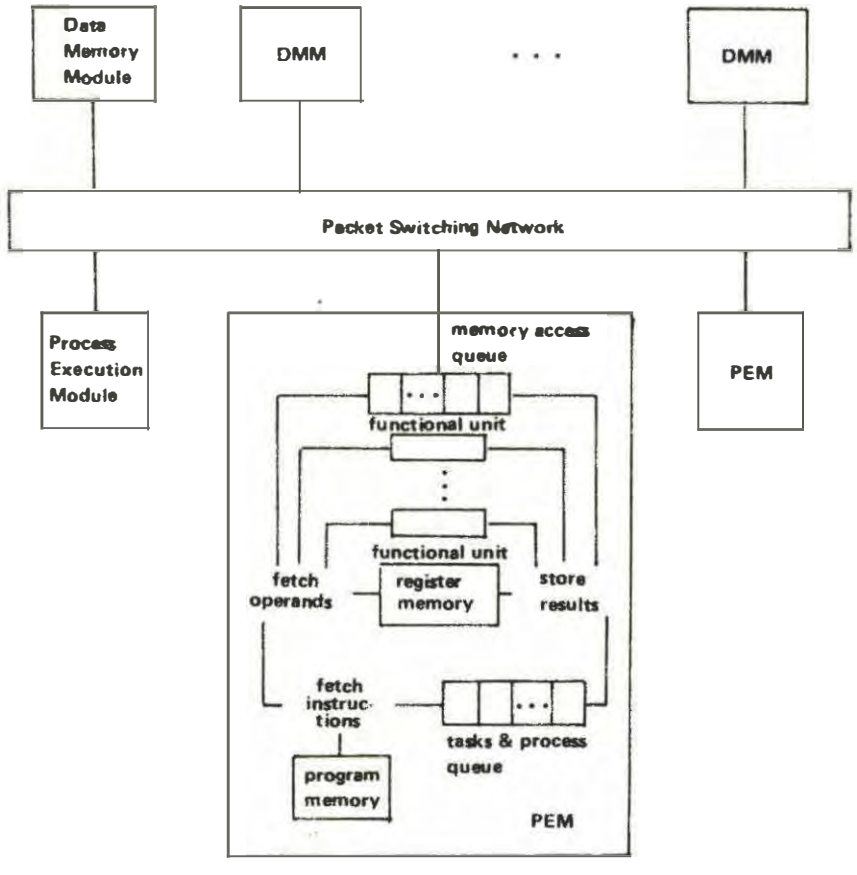

Fig. 4

The synchronization among parallel processes is done via the F/E (full/empty) bit that is tagged on special shared variables called asynchronous variables. These variables are prefixed with a "$" character. An instruction which uses the contents of an asynchronous variable must wait until the F/E bit of that variable is set to full. When the bit is set to full, the content of the variable is made available and its F/E bit is then reset to empty. This assures that no two processes can access an asynchronous variable at the same time.

The implementation on the HEP computer consists of translating the algorithms for a coordinator and for a process (see Figs. 2 and 3) into DENELCOR's Extended FORTRAN 77. Several hierarchical structures were implemented on the HEP with varying levels. Each structure was created in the main program by first requesting the number of levels in the hierarchy from the user. Then for each level, the user is asked to enter the number of children (if any). A node without any children is assumed to be a process and the rest of the nodes are coordinators.

Simulation Runs and Analysis

Several hierarchical structures were implemented on the HEP computer to measure the computation times of the algorithms. Each structure were simulated using the following run-time parameters:

a) Send emsg and ymsg messages to all the processes. In this case when a emsg or a ymsg are generated, all the processes will be in the set of destination processes. This parameter is equivalent to a DCS whereby message passing is heavy.

b) Send emsg and ymsg to a random number of processes. Here, the destination processes are selected by a random number generator. This choice of parameter simulates random message passing for a DCS.

c) Send emsg and ymsg to only one process. This parameter simulates the case where a process communicates with at most one process.

Another purpose is to find the effect of reducing the number of coordinators which subsequently reduces the number of levels in the computation of the global state. A structure with 8 processes were simulated with varying degrees of levels. The percentages of the number of signal to the number of emsg messages sent are varied to observe the effects of heavy load on the DCS.

Simulation runs

Three different tree structures with 8 processes and varying number of levels and coordinators, were implemented. The structures are shown in Figs. 5, 6 and 7. The first structure has a single coordinator controlling all 8 processes. The second structure is a binary tree with 7 coordinators and 8 processes. The third structure is one in which the level of the tree is reduced to eliminate 2 coordinators from the

previous binary tree construct. The results of the simulation runs are shown in Figs. 8-10. Comparing the different constructs, the single coordinator takes the longest time to compute for the global state. The slope of the curve varies directly as the number of signal messages goes up. Note that the receipt of a signal by a process results in the production of ymsg message. Comparing the binary tree construct and the reduced level tree, the computation times are not significantly different, although the reduced level structure takes a slightly longer computation time .

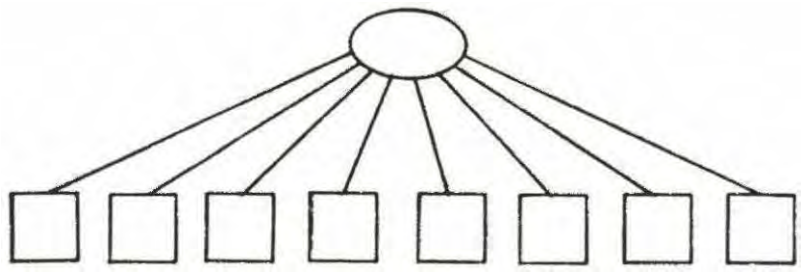

Fig. 5

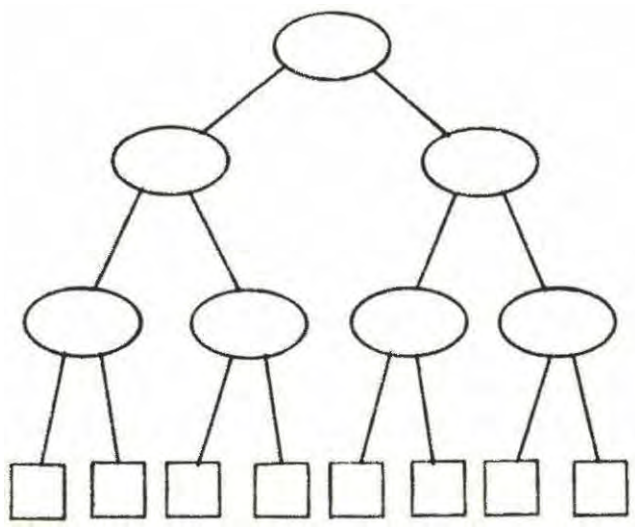

Fig. 6

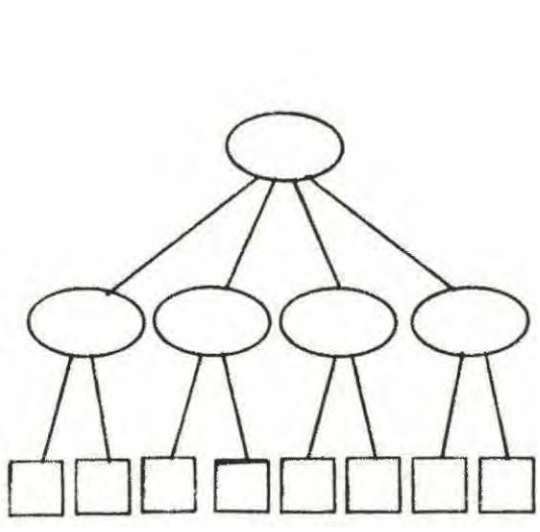

Fig. 7

Analysis of simulation runs

Shown in Figs. 8, 9 and 10 are the plot of simulation runs made for a DCS having 8 processes. Note that the simulation runs were made for 10%, 30%, 50%, 80% and 90% signal messages generated as compared to the generation of emsg messages. Five runs were made on each plot and the following are the parameters on each run:

a) Run 1 – this run has the messages ymsg and emsg sent to a random number of process(es). This run is represented in the line plot by squares.

b) Run 2 – this run has the messages ymsg and emsg sent to almost all the processes most of the time. This run is represented by '+' in the line plot.

c) Run 3 – this run has the messages ymsg and emsg sent to at most one process most of the time. This run is represented by diamonds in the line plot.

d) Run 4 – this run has the ymsg sent to the farthest process from the sending process most of the time while the emsg is sent to a random number of process(es). This run is represented by triangles.

e) Run 5 – this run has the ymsg sent to the nearest process from the sending process most of the time while the emsg is sent to a random number of process(es). This run is represented by 'x'.

Experimental Run in HEP

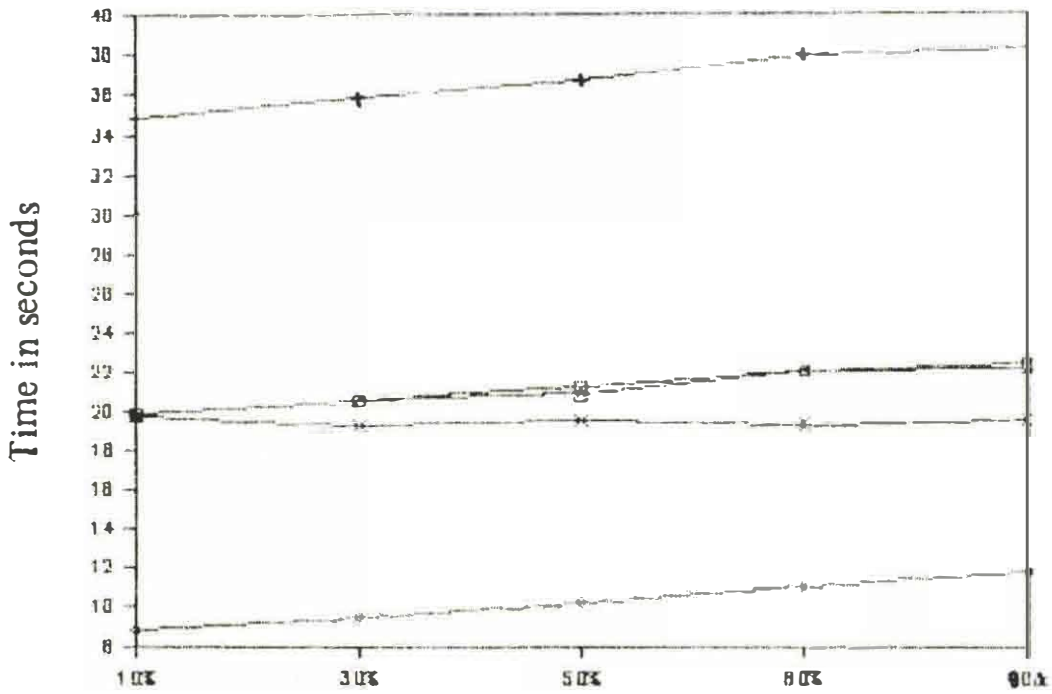

Fig. 8

Experimental Run in HEP

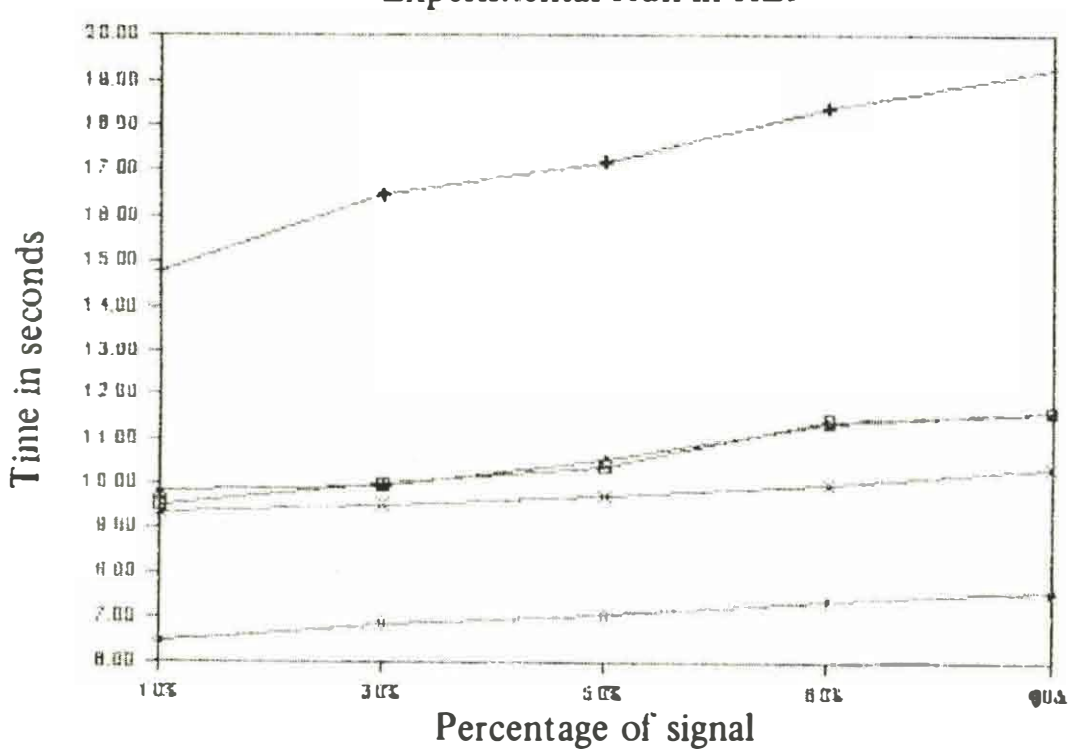

Fig. 9

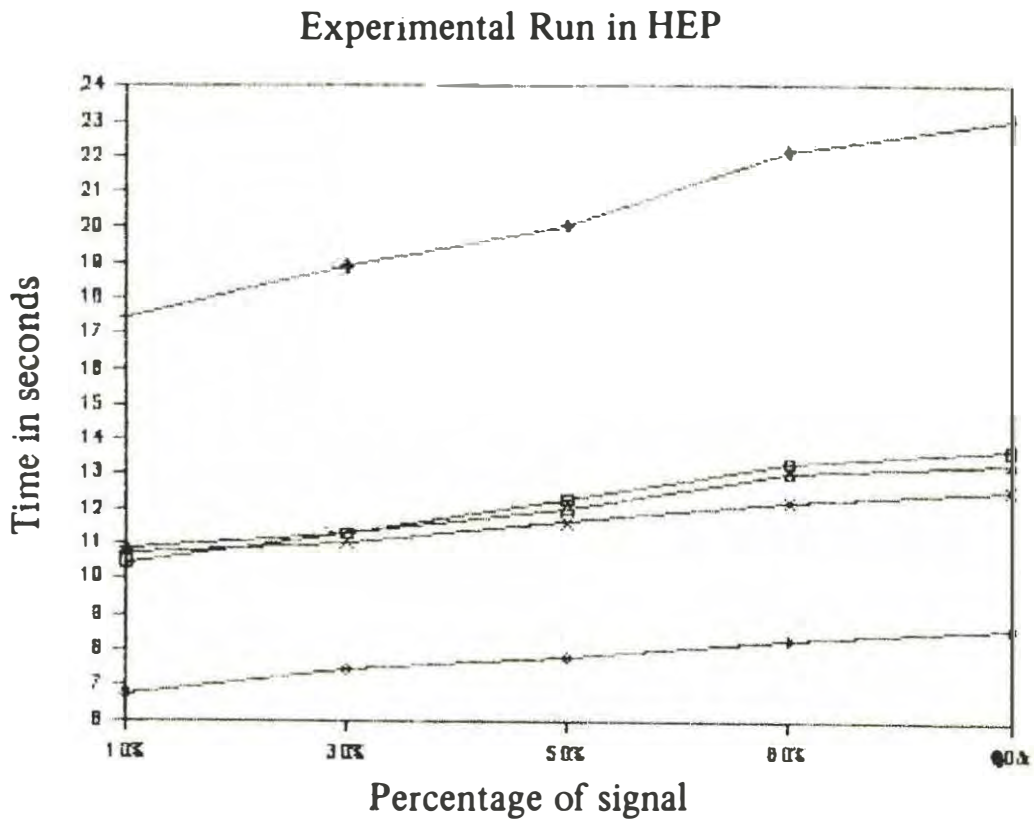

Fig. 10

Fig. 8 shows the simulation run for the construct with a single coordinator, Fig. 9 for the construct with 7 coordinators and Fig. 10 for the reduced level of hierarchy with 5 coordinators. All plots showed an increased in computation time as the percentage of signal messages generated increases. This result is expected since the receipt of signal results in the generation of ymsg. All plots also showed that Run 3 has the lowest computation time while Run 2 has the highest computation time. The reason is that Run 2 sends messages to almost all the processes while Run 3 sends a single message to just one process. Runs 1, 4 and 5 showed an average computation times as compared to the two extremes of Runs 2 and 3. The single coordinator showed the highest computation time because of bottlenecks at the single coordinator. Reducing the level of the hierarchy by one (from 7 to 5 coordinators) induces a slightly higher computation time than the construct with 7 coordinators. This indicates that having less number of levels might be equivalent in performance to the complete number of coordinators. This is a subject of further investigation.

Conclusion

This paper has proposed a hierarchy of coordinators for distributed control of a DCS. The coordinator's function is to collect information from its subordinates to compute the global state. Through the global state, optimal performance can be achieved because the complete information is available for accurate decisions. The price to pay for the computation of the global state is the synchronization and intercommunication schemes for the processes. The processes' communication are controlled by the hierarchy of coordinators. As the algorithm shows, only one

process (the imminent process) is allowed to communicate at any one time. The simulation runs have shown that the computation time varies directly as the number of levels increases (or the number of coordinators are increased).

The hierarchy of coordinators is a compromise between a centralized and a distributed approach. The centralized approach utilizes a single coordinator the failure of which results in a catastrophe for a DCS. The fully distributed approach sometimes lead to unacceptable performance because of the complexity of the computation and the overhead incurred due to roll-back procedures or re-computation of its state variables. The hierarchy of coordinators provides a structure whereby both the advantages of a central coordinator and distributed control are present. Failure of any coordinator can be handled using an electron algorithm and crash-recovery techniques.

References

[CHAN83] Chandy, K.M. et al., 1983. "Distributed Detection," *ACM Tran. on Computer Systems*, 1 (2): 144-156.

[CHAN85] Chandy, K.M. et al., 1985. "Distributed Snapshots: Determining Global Status of Distributed Systems," *ACM Tran. on Computer Systems*, 3 (1): 63-75.

[DENE84] DENELCOR, 1984. "FORTRAN 77 Reference Manual, Release 1.0," Publication No. 9008020-000, DENELCOR Inc., 17000 E. Ohio Place, Aurora, Colorado.

[GAJS85] Gajski, D.D. and J-K Peir, 1985. "Essential Issues in Multiprocessor Systems," *Computer*, 18 (6): 9-27.

[HO80] Ho, Yu-Chi, 1980. "Team Decision Theory and Information Structures," *In Proc. of the IEEE*, 68 (6): 644-654.

[HWAN84] Hwang, K. and F.A. Briggs, 1984. "Computer Architecture and Parallel Processing," *McGraw Hill Book Company*, New York, NY.

[JEFF85] Jefferson, D. and H. Sowizral, 1985. "Fast Concurrent Simulation Using the Time Warp Mechanism," *In Proc. of the 1985 Conference on Distributed Simulation*, San Diego, CA, 63-69.

[LARS82] Larson, R. E. et al., 1982. "Tutorial: Distributed Control," 2nd Ed., *IEEE Computer Society Press*, Maryland.

[MENA 79] Menasce, D. and R.R. Muntz, 1979. "Locking and Deadlock Detection and Distributed Databases," *IEEE Tran. on Software Engineering*, SE-5, (3): 195-202.

[OBER82] Obermarck, R., 1982. "Distributed Deadlock Detection Algorithm," *ACM Tran. on Databases Systems*, 7 (2): 187-208.

[STAN84] Stankovic, J.A., 1984. "A Perspective on Distributed Computer Systems," *IEEE Tran. on Computers*, C-33 (12): 1102-1115.