Null Messages Cancellation Through Load Balancing in Distributed Simulations

Azzedine Boukerche and Sajal K. Das

Department of Computer Sciences, University of North Texas, Denton, TX. USA

Abstract. This paper presents the results of an emperical study of the effects of null messages cancellation through load balancing in distributed simulations. Our null-message-cancellation scheme is a modification of the Chandy-Misra protocol, wherein old null messages are discarded. We propose two load balancing strategies based upon a process migration and study its scalability on an Intel paragon machine. The experimental results show the impact of our load balancing schemes on the null message concellation protocol, where a significant reduction of the null-message overhead was observed.

1 Introduction

Due to its importance, load balancing is a well studied problem in parallel and distributed systems in general, and a significant body of literature exist. Most of the research focused on designing effective partitioning and load balancing algorithms with as low overhead as possible. The methods proposed range from analytical study plus simulation to implementation plus testing. However, most of the load balancing algorithms studied in distributed systems are not readily suitable for distributed simulations. This is due the causality and the synchronization constraints[1] that exacerbate dependencies between the logical processes.

In this paper, we make use of Chandy-Misra protocol [4] which employs null messages in order to avoid deadlocks and to increase the parallelism of the simulation. When an event is sent on an output link a null message bearing the same time stamp as the event message is sent on all other output links. As is well known, it is possible to generate an inordinate number of null messages under this scheme, nullifying any performance gain [1,2]. To increase the efficiency of this basic scheme, we employ the following cancellation approach. In the event that a null message is queued at an LP and a subsequent message arrives on the same channel, we overwrite the old null message with the new message. We associate one buffer with each input channel at an LP to store null messages, thereby saving space and the time required to perform the queuing and dequeuing operations associated with null messages.

---

[1] Recall that in the conservative approach, a logical (LP) process is not allowed to process an event unless it is certain that it will not receive an earlier event message.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 562–569, 1999.

2 Null-Message Cancellation Through Load Balancing

We propose to implement the load balancing facility as two kinds of processes: *load balancer* and *process migration* processes. In our implementation, both processes belong to the same processor[2]. A load balancer makes decision on *when* to move *which* process to *where*, while migration process carries out the decision made by the load balancer to move processes between processors. In other words, the load balancer plays the policy role while the migration process supplies the mechanism in the load balancing facilities. This separation between policy and migration provides us a flexibility of various load balancing schemes such as preemptive and non-preemptive strategies [1]. Several approaches to both information gathering and decision making were investigated. We propose to use a centralized and a multi-level load balancing strategy due mainly to reduce the message traffic over a fully distributed approach.

- **Centralized Load Balancing (CL)**: The load balancer sends periodically a message $< Request\_Load >$ to each processor (Pr_k) requesting the load of each logical process, LP_i, which is assigned to Pr_k. When all processors have responded, the load balancer computes the new load balance. later, we will define the load, and show how to compute the load. Once the load balancer makes the decision on *when to move which LP to where*, it sends $< Migrate_{Request}, Which : LP, Where\_To : Processor >$ to the migration process which in turns initiates the migration mechanism. Note that the performance of this approach can be degraded significantly by the fact that all request/reply messages are routed through the LBF. Hence, this scheme may lead to a major bottleneck.

- **Multi-Level Load Balancing (ML)**: In this approach, the goal is to prevent the bottleneck and reduce the message traffic over a fully distributed approach. Several hierarchical strategies were investigated, in which processors are grouped and work loads among processors are balanced hierarchically through multiple levels. For simplification, we settle with a Two-Level scheme. In level 1, we use a centralized approach that makes use of a *(Global) Load Balancing Facility, (GLBF)*, and the global state consisting of process/processors mapping table. The load balancer ($GLBF$) sends periodically a message $< Request\_Load >$ to a specific processor, called $First\_Processor$, within each cluster requesting the average load of all processors within that cluster.

In level 2, the processors are partitioned into clusters. The processors within each cluster are structured as a virtual ring and operate in parallel to collect the work loads of processors. We choose to use a distributed scheme at level 2 to improve the efficiency and the performance of our load balancing algorithm. A virtual ring is designed to be traversed by a token, which originates at a particular processor, called *First_Processor*, passes through intermediate processors, and ends its traversal at a preidentified processor called *Last_Processor*. Each of the rings will have its own circulating token, so that information (i.e., work loads) gathering within the rings is concurrent. As the token travels through the processors of the ring, it accumulates such information as the load of each

---

[2] One could as well use two processors instead.

processors and *id* of the processors that contain the highest/lowest loads, so that when it arrives at the *last_processor*, information have been gathered from all processors of the ring. The token is generated at *First_Processor* whenever the *first_processor* receives a *Request_Load* message from the $GLBF$. It is destroyed at the *Last_Processor*. However, since the *Last_Processor* possess the information requested by the $GLBF$, *Last_Processor* sends a reply message with the updated information about the ring to $GLBF$. When the *Last_Processor* (s) of each cluster have responded, the load balancer ($GLBF$) computes the new load balance. Once the load balancer makes the decision on *when* to move *which* LP to *where*, it sends $< Migrate_{Request}, Which : LP, Where\_To : Processor >$ to the migration process which in turns initiates the migration mechanism.

Process Migration: A process migration facility in a distributed system dynamically relocates running processes among the component machines. Such relocation can help cope up with dynamic fluctuations in load and service needs. The use of process migration for optimistic distributed simulation may also be found in [6]. We choose the following implementation for the process migration. Let us denote by $MAP(LP_i)$ the identity of the processor handling the logical process LP_i where each processor maintains the mapping table. We denote by S_{LPi} and D_{LP_i}, the source and the destination processors, which respectively indicate from *where* to move LP_i to *where*. In our mechanism, when LP_i migrates from processor S_{LP_i} to D_{LP_i}, the process transfer occurs in three phases:

i) *Establishment:* Processors S_{LP_i} and D_{LP_i}, after being told by the LBF (i.e., the process migration) to migrate the logical process LP_i, agree to the transfer. D_{LP_i} will keep all messages that should be sent to LP_i (hence to S_{LP_i}). That is possible only if LP_i has neighbors that are assigned to D_{LP_i}. If D_{LP_i} does not keep these messages, they would have been sent to S_{LP_i} which will forward them back to D_{LP_i} since LP_i does not reside anymore in S_{LP_i}. In order to avoid this, we decided to keep these messages and then forward them to LP_i which resides now in Processor D_{LP_i}. When the migration of LP_i from S_{LP_i} to D_{LP_i} is terminated, D_{LP_i} will forward these messages to LP_i.

ii) *Transfer:*S_{LP_i} sends a message of type *migrate*, i.e.,$< Migrate_{LP}, Which: LP_i, Where\_To : D_{LP_i} >$ to D_{LP_i}. This message will include all information regarding LP_i, i.e., data structure pertaining to the migrant process including its local simulation time (LST), and the event messages waiting in its input queues. Note that the transfer is made only when a process finishes processing its event message (and not while it is processing the event).

iii) *Notification:* S_{LP_i} must inform its neighboring processors about LP_i-migration. Upon receipt of such message, each processor will update its mapping table, MAP. In our implementation, S_{LP_i} will notify it's neighbors only after the migration procedure is finished. This will avoid any confusion, such as having a node being notified about the transfer before actually performing the migration.

Once S_{LP_i} has sent the message of type *migrate*, any messages received later by S_{LP_i} and scheduled for LP_i will be forwarded to D_{LP_i}. In order to decrease the communication overhead, we store the messages sent to LP_i and check periodically if S_{LP_i} receives messages for LP_i, which will be forwarded to

D_{LP_i}. Our experiments show that this simple mechanism decreases significantly the network traffic and the communication overhead.

Load Calculation: The computational load in our distributed simulation model consists of executing the null and the real messages. We choose to measure the current load at each processor and then balance the load at run time so that all available processors will be assigned the same load.

Let R^k_{avg} and N^k_{avg} the average CPU-queue lengths for real messages and null messages at each processor Pr_k. In order to distribute the null messages and the real messages among all all the processors, let us define the (normalized) load at each processor Pr_k as a function, $\mathcal{F}$, of R^k_{avg} and N^k_{avg}. It is defined as follows : $Load_k = \mathcal{F}(R^k_{avg}, N^k_{avg}) = \alpha R^k_{avg}/R_{avg} + (1-\alpha)N^k_{avg}/N_{avg}$; where α and $(1-\alpha)$ are respectively the relative weights of the corresponding parameters.

The value of α was determined empirically. In our experiments, $\alpha = 0.25$ yielded good results. An intuitive explanation lies in the nature of the Chandy-Misra protocol and the cancellation scheme described at the beginning of this section. Since the Chandy-Misra protocol generates an inordinate number of null messages, the success and the efficiency of our scheme depends on the number of null messages present during the simulation. The more the null messages, the more likely that the scheme will overwrite the old null messages, thereby reducing the null messages that would have been generated.

The load is well balanced if $Load_k$ is equal to 1, i.e., $\alpha + (1-\alpha)$. If the load balancing algorithm requires that each processor strives to adjust its load to be exactly equal to the average, processor *thrashing* may result. The migration of a process to an underloaded processor may increase its load to the point of making it overloaded, necessitating the migration of that LP to yet another processor. We avoid this by defining a *tolerance* factor, δ, within which the load of all the processors should fall. The value of δ is to be determined empirically. (In our experiments, $\delta = 0.2$ yielded good results.) Thus, if we suppose that the maximum deviation δ from an equal distribution is allowed in the simulation, then $1 - \delta \leq Load_k \leq 1 + \delta, \quad$ where $k = 1, ..., K$, and K is the number of processors. Consequently, the processor is overloaded if $Load_k > 1 + \delta$. The tolerance factor determines the responsiveness as well as the communication cost of the load balancing algorithm. However, a wider tolerance allows more variation in load among the processors. Choosing the appropriate tolerance allows the load balancing algorithm to adapt in the distributed simulation environment. Our approach is then to balance the workload among the processors while minimizing the inter-processor communication. One way to decrease the communication overhead is to combine logical processes together. So we choose the following method: The load balancer selects a (heavily overloaded) process LP_s from the heavily overloaded processor, such that LP_s has at least r neighbor processes $\{LP_1, LP_2, ..., LP_r\}$, and has at least one process LP_i, where $i = 1...r$, assigned to a different processor than the one containing LP_s. Here r is can be determined either empirically or from the network topology under consideration Then, the migration process sends this selected process to the lightly underloaded neighboring processor.

3 Simulation Experiments

The experiments were conducted on an Intel Paragon at CalTech. The Paragon is a distributed memory multicomputer, consisting of 72 nodes, arranged in a two-dimensional mesh. In this paper, we consider a distributed communication network, and a traffic flow network. The communication and the traffic flow networks are realistic simulation models that mimic many real world problems.

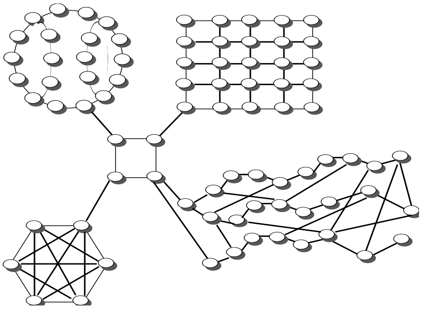


Fig. 1. Distributed Communication Network

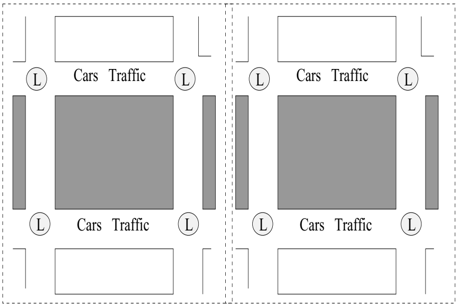


Fig. 2. Traffic Flow Network

An initial set of experiments was performed in which the routing probability was characterized by a uniform distribution, In this case, the obtained results indicated that our dynamic load balancing did not provide any significant improvement over a static partitioning algorithm. These results led us to ask if these results are applicable to *asymmetric* workload as well. Consequently, we selected an unbalanced network with the following routing probability: at the beginning of each simulation run, each process randomly selects one link as a *favorite* link which will receive twice as many messages as the other(s).

The Distributed communication model [6] used in our experiments, models a national network consisting of four regions which are connected through four centrally located delays. Final message destination is not known when a message is created. Hence, no routing algorithm is simulated. Instead one third of messages arrival are forwarded to neighboring nodes. A uniform distribution is employed to select which neighbor receives a message. Messages may flow between any two nodes, possibly through several paths. Nodes receive messages at varying rates that are dependent on traffic patterns. There are numerous deadlocks in this model, and hence it provides a stress test for any conservative synchronization mechanism. Various simulation conditions were created by mimicking the daily load fluctuation found in large communication network operating across time zones. Therefore, in the pipeline region, for instance, we arranged the sub-region into stages, and processes in the same stages perform the same normal distribution with a standard deviation 20%. The time required to execute a message is significantly greater than the time to raise an event message in the next stage (message passing delay). We use a pipeline sub-model with 26 processes, a toroid sub-model with 25 processes, a fully connected sub-model with 5 processes, and a circular sub-model with 20 processes. We choose a shifted exponential service time distribution for the 3 sub-regions (circular, toroid, and fully connected sub-regions).

In our next model, we represent the traffic network as a square mesh composed of streets running in the horizontal and vertical directions with traffic lights at the intersections of the street. This model is partitioned into sub-systems; which we refer to as grids. Each grid is assigned to a logical process (LP). The cars enter the simulation and travel within and between the grids using message passing. To reflect traffic flow network model, we define a light interval to be a triple $< r, g, y >$; where r is the number of clock ticks that the light is red, g is the number of clock ticks that the light is green, and y is the number of clock ticks that the light is yellow. Fig. 2 illustrates an example of network traffic with two grids, where each grid contains four lights. A car enters the simulation at a construction site labeled a *Source_Sink*. All of the boundary streets, as shown in this figure, are sources and sinks where cars are generated according to a probability distribution. A car might travel either North, South, West, or East with a fixed probability distribution of changing directions at any given intersection. We also consider the contention problem at the intersection, i.e., if a car is turning left into a path of a car going straight, then a contention mechanism will inhibit one of the cars until the other car clears the intersection. In our experiments, we choose an average traffic flow of 100,000 cars. The number of lights is the traffic network is held constant at 576 lights for controlling the workload of the system. To make the simulation more interesting, we choose 20 hot spots uniformly distributed among all the grids, where the probability of car is generated at a source is 0.25.

- **Performance Results**

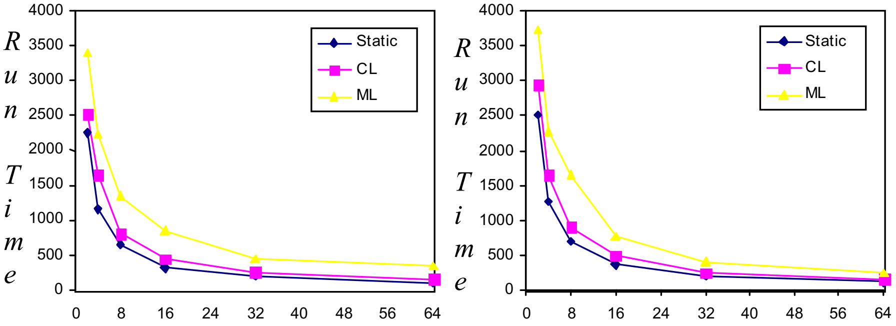
The experimental results were obtained by averaging several trial runs. First, we present in the form of graphs the results for the execution time (in seconds). Next, we study the synchronization overhead as *the null message ratio* (NMR) which is defined as the number of null messages processed by the simulation using Chandy-Misra null-message approach divided by the number of real messages processed. It is important to understand that there are no existing (dynamic) load balancing algorithms for the conservative simulation paradigm against which we can compare the performance our algorithm. Hence comparisons were made to a static partitioning algorithm. The static algorithm used for comparisons is based upon [5]. The algorithm basically attempts to minimize the communication overhead by uniformly distributing the execution load among the processors. We assume that there are as many clusters as there are processors, and each cluster is assigned to one processor. The algorithm starts with an initial random partition, and then iteratively moves processes between clusters until no improvements can be found (a local optimum). The processes are moved among the clusters so that the total cut-size is reduced and the cluster sizes remain balanced. All possible moves for each process are considered, and the process which contributes to the maximum gain is chosen. A process is moved only if it does not violate the cluster size constraints.

Let us now turn to our results. Fig. 3 show respectively the run times for distributed communication network and flow traffic models. The trends of the curves and the relative performances are relatively similar to the other simulation models. We see a reduction of 20-25% in the execution time when we increase the number of processors from 2 to 8, and 40-45% when we increase the number of processors from 16 to 64. We observe a significant decrease in the running time when the CL and ML load balancing schemes are used, for both methods and for both models. We observe 50% reduction in the execution time when ML strategy is used compared to the static algorithm. The results also indicate that the CL scheduling performs worse than the ML load balancing method for all four simulation workload models. This is due to the nature of the centralized scheme that leads to a major bottleneck, thereby slowing down the simulation. On the other hand, the ML scheduling tries to avoid the bottleneck and reduces the message traffic by undertaking a fully distributed approach.

We now examine the synchronization overhead,, namely the null-message ratio (NMR). Fig. 4 displays the NMR as a function of the number of processors employed in the network models. We observe a significant reduction of null-messages for all load balancing schemes in all four models. Also NMR increases as the number of processors increases for all four simulation models. For instance, in Fig. 8, if we confine ourselves to less than 8 processors, an approximately 20-25% reduction of the synchronization overhead using the CL dynamic load balancing algorithm is observed over the static one. Increasing the number of processors from 8 to 32, we observe about 30-40% reduction of NMR. The hierarchical scheme strategy always outperforms the centralized one. We observe a 45-50% reduction using ML strategy over the static one, when we increase the number of of processor from 32 to 64. Similar results were obtained with the fully connected communication model, the distributed communication network, and the traffic flow model. The trends of the curves and the relative performances are relatively similar. These results conclude that the multi-level load balancing strategy significantly reduces the synchronization overhead when compared to a centralized strategy. In other words, a careful dynamic load balancing improves the performance of a conservative distributed simulation.

4 Conclusions

In this paper, we have presented a null-messages cancellation scheme through load balancing in distributed simulation. Our results show that a significant reduction of null message using our scheme. A significant decrease in run-time was also obtained with the use of our proposed cancellation scheme as compared to the use of a static partitioning algorithm. We note that a hierarchical approach seems to be a promising solution in reducing further the execution time of the simulation.



(a) Distributed Communication Network *(b) Traffic Flow Control*

Fig. 3. *Run-Time Vs. Number of Processors*

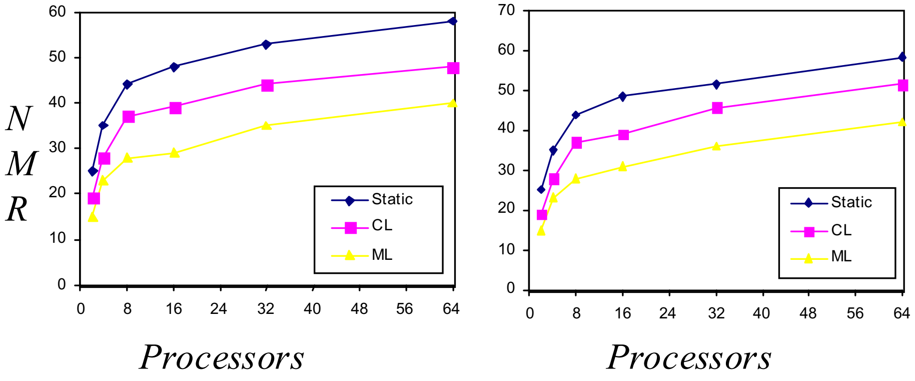


Fig. 4. *Synchronization Overhead Vs. Number of Processors*

References

[1] Boukerche, A.: “Time Management in Parallel Simulation”, in High Performance Cluster Computing, Vol. 2, Prentice Hall, 1999. (Eds. R. Buyya, and M. Baker).

[2] Boukerche, A., and Tropper C.: “A Static Partitioning and Mapping Algorithm for Conservative Parallel Simulations”, IEEE/ACM PADS’94, 1994, 164–172.

[3] Fujimoto, R. M.: “Parallel Discrete Event Simulation”, CACM, 33(10) 1990.

[4] Misra, J., “Distributed Discrete-Event Simulation”, ACM *Comp. Surveys*, 1986.

[5] Nandy, B., and Loucks, W. M., “An Algorithm for Partitioning and Mapping Conservative Parallel Simulation onto Multicomputer”, PADS’92, 139–146.

[6] Glazer, D., and Tropper, C.,“On Process Migration and Load Balancing in Time Warp”, *IEEE Trans. on Parallel and Distributed Systems*, 4(3), 1993, 318–327.