An Evaluation of Conservative Protocols for Bulk-Synchronous Parallel Discrete-Event Simulation

Mauricio Marín
Computing Department, University of Magellan
Casilla 113-D, Punta Arenas, Chile
E-mail: mmarin@ona.fi.umag.cl

Abstract

In this paper we present an empirical comparison among conservative synchronization protocols for distributed discrete-event simulation which are suitable for the bulk-synchronous parallel (BSP) model of computing. A number of optimizations were introduced which produced improved protocols. We also establish comparisons with an optimistic synchronization protocol called BSP Time Warp. We assessed the performance of the protocols by using architecture independent performance metrics, and quantified empirically these metrics by executing a demanding work-load under different situations.

1 Introduction

In many large scale systems parallel discrete-event simulation (PDES) [2] is the only form of computation that is capable of achieving reasonable running times. Parallelism is introduced by partitioning the system into a set of concurrent objects called logical processes (LPs). Events take place within LPs, their effect is the change of LP states, and LPs may schedule the occurrence of events in other LPs by sending event messages to them. As the LPs are placed on different processors, a synchronization protocol is required to ensure that events scheduled by different processors on any LP are processed in chronological order.

In conservative protocols [6] events are allowed to take place when there is certainty that no earlier events can take place in the respective LPs. This poses a difficult problem since determining when it is "safe" to process a given event requires information of the behavior of other LPs that might schedule events in the same place. This is in contrast with the approach adopted by optimistic protocols [3, 2] which simply processes available events and performs corrections whenever causality errors take place. However, the reward is that state saving and roll-back procedures are not longer required and this fact can potentially reduce memory and running time requirements.

Conservative simulation has its shortcomings as well. In asynchronous protocols the LPs block themselves when there is insufficient information to determine whether the available events are safe or not. This leads to the problem of deadlock occurrences which introduces new overheads associated with their detection and recovery. Also the need for keeping information about what events the LPs may schedule in others, leads one to deal with the particular communication topology of the LPs. This makes these protocols less useful in simulations where the communication topology changes dynamically or the fan-in/fan-out of the communication is large (e.g., all to all).

On the other hand, synchronous protocols base their operation upon global information in a manner that does not require special consideration of the communication topology. However, this "global information" by itself exacerbates the requirements of global synchronization in real and simulation time which can degrade performance significantly. In addition, these protocols need to use more knowledge about the specific operation of the simulation model which in turn complicates their utilization as general purpose simulation environments.

Conservative protocols promote a determined synchronization strategy which may have quite different realizations in algorithmic terms depending on the model of parallel computing. The main contribution of this paper is the proposal of efficient algorithms for conservative synchronization protocols on the BSP model of parallel computing [10], and the evaluation of their comparative performance.

The conservative protocols considered are Time Buckets (TB) [9], Chandy-Misra-Bryant (CMB) [6], Bounded Lag (BL) [4], Conservative Time Windows (CTW) [1], and YAWNS [7].

BSP is a model in which both computation and communication take place in bulk before the next point of global synchronization of processors. A BSP program is composed of a sequence of *supersteps*. During each superstep, the processors may only perform computations on data held in their local memories and/or send messages to other processors. These messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronization of the processors. Note that the BSP model can be easily implemented in today's parallel architectures as facilities for performing message sending and barrier synchronization are commonly available.

In this paper we evaluate synchronous and asynchronous protocols and, unlike previous comparative studies, we compare these protocols under the same model of parallel computing, namely the BSP model (our study also includes protocols which have not been compared so far in the literature). Synchronous protocols have an almost straightforward implementation in BSP because of their bulk-synchronous operation. Asynchronous protocols, on the other hand, with roots in either asynchronous message passing or shared memory systems, require a radically different algorithmic design in order to avoid the overheads inherited by the simple mapping from one model of computing to the another.

The proposed algorithms, whose BSP features enable us to exploit some useful optimizations such as reduction of message traffic, can be considered as new realizations of the synchronization strategies supported by the respective message passing or shared memory algorithms.

Our quantitative results are mainly empirical and, as is the usual practice in this kind of study, they are restricted to a particular (though demanding) work-load. However, these results in combination with existing knowledge on performance of synchronization protocols enable us to draw qualitative conclusions which provide a more general view of the comparative performance of the protocols.

2 The work-load

To evaluate the performance of the different protocols studied in this paper, we use a work-load which is regarded as a demanding benchmark for conservative synchronization protocols since it contains multiple feedback loops. We refer to the simulation of a non-preemptive single-server queuing network with FCFS discipline. The servers (LPs) are organized as an $n \times n$ toroidal grid so that

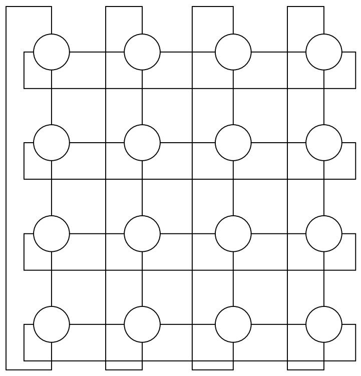


Figure 1: Toroidal topology with wrap around links.

id	P	D_p	D	Loc
1	16x16	2x2	2	1/2
2	8x8	4x4	2	1/4
3	4x4	8x8	2	1/8
4	2x2	16x16	2	1/16
5	16x16	2x2	8	1/2
6	8x8	4x4	8	1/4
7	4x4	8x8	8	1/8
8	2x2	16x16	8	1/16

Table 1: Queuing network instances.

each node (server) has four neighboring nodes, see figure 1. The communication between two neighboring nodes is bi-directional, i.e., they may send each other messages reporting the arrival of "jobs" to the respective queues. A fixed number of jobs is kept flowing constantly throughout the network by defining an initial number of jobs in each server and routing jobs uniformly at random among the four neighbors. There are three basic events in the simulation model: job-arrival (ARRIVAL), begin-service (SERVICE) and service-completion (COMPLETION). The sequential simulation of this queuing model is described in figure 2.

The parallel simulation is performed on P processors with D_p queuing servers per processor. The empirical data shown in this paper were obtained from eight instances of the queuing network, see table 1. This table describes cases for $D_p P = 1024$ with servers under low ($D = 2$) and high ($D = 8$) load. The parameter D is the number of jobs per server at the start of the simulation. In the table, the column Loc is the locality present in each instance of the model, which we define as the fraction of messages sent to other processors.

The service times of the queuing servers are the

```
/* Define: $e$= event type, $i$= source server,
        $t$= event time, $j$= destination server */
while (not-end-sim())
    NextEvent($e, i, t$);
    case event $e$ is
        ARRIVAL:
            Jobs[$i$]:= Jobs[$i$]+1;
            if Server[$i$] = IDLE then
                goto SERVICE;
        SERVICE:
            Jobs[$i$]:= Jobs[$i$]-1;
            Server[$i$]:= BUSY;
            $t$:= $t$ + service-time($i$);
            Schedule($t$,COMPLETION,$i$);
            $j$:= Neighbor($i$,random(1,4));
            Schedule($t$,ARRIVAL,$j$);
        COMPLETION:
            if Jobs[$i$] = 0 then Server[$i$]:= IDLE;
            else goto SERVICE;
    endcase
endwhile
```

Figure 2: Sequential simulation.

sum of a constant (base lookahead) $L_A \geq 0.1$ and a continuous random variable exponentially distributed with mean one. Also note that the pseudocode in the figure 2 is designed so that *pre-sending* of jobs is exploited [7]. That is, since jobs that are receiving service cannot be preempted, both completion and arrival events are scheduled at the same time. This feature, which in the sequential simulation is equivalent to scheduling an arrival event right after its associated completion event, increases the available parallelism of the model since completion and arrival events can be processed in parallel (i.e., job pre-sending increases lookahead).

3 Conservative protocols

The reader is referred to [5] for a detailed explanation of the BSP implementation of the protocols studied in this paper. We only give here a brief description of their operation.

The TB protocol. Since we have defined service times such that their values are at least the constant L_A, the straightforward method of parallel simulation consists of defining a moving time window of width L_A [9]. The window is defined as $[t, t + L_A)$ where t is equal to the simulation *floor*, i.e., the time of the next event in the whole system. Only the events e with time $e.t < t + L_A$ are simulated in each event processing superstep. Between event processing supersteps, a min-reduction is executed in order to calculate the simulation floor for the next cycle. Note that job pre-sending is compulsory in this case since sending arrival events right after their respective completion events produces $L_A = 0$.

The CMB-NM protocol. Each LP has a set of input message channels and a set of output message channels wherein messages are received/sent from/to other LPs. These channels are defined in accordance with the communication topology of the LPs. Each LP selects for processing the event with the least time from its input queues provided that *all* these queues are not empty. Special null-messages are used to communicate lower bounds on the time of the next event to arrive at empty input queues. In our case, the time of a null message is either the time of the current completion event plus L_A, or the least null message time plus L_A if the server is idling. The null messages are only sent between LPs which are located in different processors. It suffices to send just one null-message between these LPs at the end of each superstep; the one with the largest time (per LP pair). In addition, the sending of a null message can be avoided if its timestamp is not greater than the timestamp of the previous null-message sent through the same channel. The combination of both schemes reduces significantly the null message traffic among processors.

The BL protocol. It works in a bulk-synchronous fashion. BL is based on the concept of *minimum propagation delay* $d(i, j)$ which is defined as the minimum time increment of events scheduled from LP i to LP j (e.g., if an event e_i with time $e_i.t$ takes place in LP i and e_i schedules event e_j in LP j, then $e_j.t \geq e_i.t + d(i, j)$). For LPs i and j which do not have an explicit communication link between them, the quantity $d(i, j)$ is the shortest path from i to j. For a given cycle, the occurrence of an event e_i at LP i is simulated iff

$$e_i.t < \alpha(i) = \min_{j \neq i}\{d(j, i) + \min\{T_j, d(i, j) + T_i\}\}$$

where T_j and T_i are the times of the next event in LPs j and i at the start of the current cycle. However, testing every LP $j \neq i$ takes $O(n)$ time, with n being the total number of LPs, and thereby only the LPs located within a pre-defined neighborhood are considered. To this end, a user-defined time bound B is used to define the extent of the neighborhood: in calculating the above $\alpha(i)$ quantity, we only consider the subset of LPs j such that $d(j, i) \leq B$.

The CWT-SP protocol. It works like CMB but determines a time window $[T_i, U_i)$ for each LP i with T_i being the time of the next event in i and U_i being a lower bound on the time in which any other LP in the system can schedule an event in i. A single-source-all-shortest-paths algorithm is used to calculate U_i. Null messages are employed to spread out among the processors the times of the next events in each LP.

The YAWNS protocol. It does not require explicit consideration of the communication topology among LPs. However, it is based on more restrictive assumptions about the features of the system being simulated: job pre-sending and the times of the next events to be scheduled in other LPs. These two factors increase the lookahead of the simulation. The operation of YAWNS is as follows. Let us define $\alpha(i)$ as the departure time of the *next* job to receive service in a non-preemptive server (LP) i (i.e., we exclude the job currently in service). Every time we process a job-arrival event in the LP i, we pre-compute its future service time s_i, so that at any instant t we can calculate $\alpha(i)$ as $\alpha(i) = t + \max\{r_i, a_i - t\} + s_i$, where r_i is the residual time of the job receiving service at t and a_i is the arrival time of the next job to receive service [7]. In addition, every time that a job starts receiving service, an event message indicating the arrival of the job to its next server is sent immediately to the respective LP (job pre-sending). In this way, if the whole simulation has been barrier synchronized at simulation time t, then a new cycle can be initiated by processing events e with time $e.t$ satisfying

$$e.t < \min_{\text{all servers } i} \{\alpha(i)\}\,.$$

These events are safe since the minimum $\alpha(i)$ is actually the minimum time in which any LP can be affected by other LP.

The BSP TW protocol. This is an optimistic protocol and thereby it makes use of a roll-back mechanism which re-starts the simulation of a given LP when an out-of-sequence event is detected. In each superstep, each processor simulates the occurrence of events in accordance with messages (events) received at the start of the current and/or previous superstep(s). This simulation is effected sequentially by using a processor event-list. In addition, for each processor, there is an (adaptive) upper limit to the number of events allowed to take place in each superstep. Newly-generated and rolled-back events are treated identically: they are all kept in their respective processor event-lists so that they are selected for processing in chronological timestamp order. The upper limits to events are calculated automatically by an adaptive algorithm [5].

4 Speedups Analysis

The results presented in this section were obtained by executing the sequential simulator of BSP protocols. From these executions, the only real-time measure we collected from was the total running time required by the sequential BSP simulator to complete a given simulation. This produced measures of the software overheads involved in the proposed BSP implementation of the protocols (these overheads are indeed important to obtain as they are not present in the sequential simulation of the same system, and thereby they can affect the speedup noticeably). The other measures came from counters located at different points of the BSP simulator (counters that were removed from the simulator while measuring actual running time). In the following we make use of the BSP parameters g and l to predict speedup from the simulation data with l being the cost of barrier synchronizing the processors and g the cost of sending one-word message through the communication network. Our aim is to study the qualitative effects that various scenarios have on speedup under a situation that is favorable to conservative protocols. This because we also introduce an efficient optimistic protocol — BSP Time Warp (TW) [5] — in these comparisons.

It is not difficult to see that in order to achieve reasonable performance on a machine with high l and g we need to look at protocols with low requirements of synchronization and communication. In this case, software overheads are a less crucial issue in selecting the synchronization protocol. On the other hand, simulations on a machine with very low l and g make software overheads the most important issue in selecting the synchronization protocol. The extreme case would be to assume $l = g = 0$ which is rather unrealistic. Instead we use in this section the l and g values of a machine with very low cost of synchronization and communication but also with slow processors as compared to the SGI PowerChallenge's processors were the sequential BSP simulations were executed. This provides proper emphasis to software overheads while l and g are given real-life values.

We refer to the Cray T3D with 256 processors with average performance of 12 Mflops per processor, l values ranging from 33 μs to 12 μs, and g values from 0.19 μs/word to 0.08 μs/word. These values were obtained by running benchmark BSP programs on the machine [8]. The SGI machine has an average performance of 72 Mflops so we amplified the SGI running time measures by a factor of 72/12=6. The cost of processing each event in the sequential simulation (not the sequential BSP simulation) of the queuing network is $\approx 10\ \mu$s (SGI). Thus the event granularity in the Cray T3D is good since it is about twice the cost of barrier synchronizing the processors.

The first set of predicted speedups is shown in figures 3.a, b and c. These figures show speedups for the eight instances of the work-load. The data are represented as a collection of segments delimited by two points. For each segment the first point (left to right) is the speedup value for low load $D = 2$ whereas the second point is speedup for high load $D = 8$. For each protocol there are four segments representing speedups for P= 4, 16, 64 and 256 processors (bottom to top). The protocols are identified in the x-axis of each figure and they are TB, CMB-NM (NM), BL, CTW-SP (CW), BSP TW (TW), BSP TW without job pre-sending (TW2), CMB-NM with YAWNS lookahead (NM2), CTW-SP with YAWNS lookahead (CW2), and YAWNS (YWS). The y-axis indicates the specific speedup values associated with the segment points.

Figure 3.a shows speedup values for small lookahead $L_A = 0.1$. In most cases the best speedups are achieved by TW (BSP TW) and NM2 (CMB-NM with YAWNS lookahead). For high load $D = 8$, NM2 outperforms TW for $P \leq 64$. In particular, NM2 achieves speedup $> P$ as a result of sequential BSP simulator running times less than the respective sequential simulation running times. This shows that the synchronization and communication costs are indeed small in this case study (our data indicate that the cost of synchronization is less than 10% and communication is less than 2% of the total running time). Note that in all the following figures, except figure 3.c, we assume that the cost of min-reductions is zero.

Figure 3.b shows a different situation in terms of comparative performance among the protocols. Here the base lookahead is very large $L_A = 1$ (equal to the mean value of the exponential distribution). In this case the total number of supersteps developed by each protocol is fairly the same (the experimental data show differences within a factor of 2). Given the small proportion in which the cost of communication participates in the total running time of each protocol, the figure 3.b describes what happens in a situation in which software overheads are the dominant factor in the speedups. The figure shows that synchronous protocols with low software overheads and small requirements in commu-

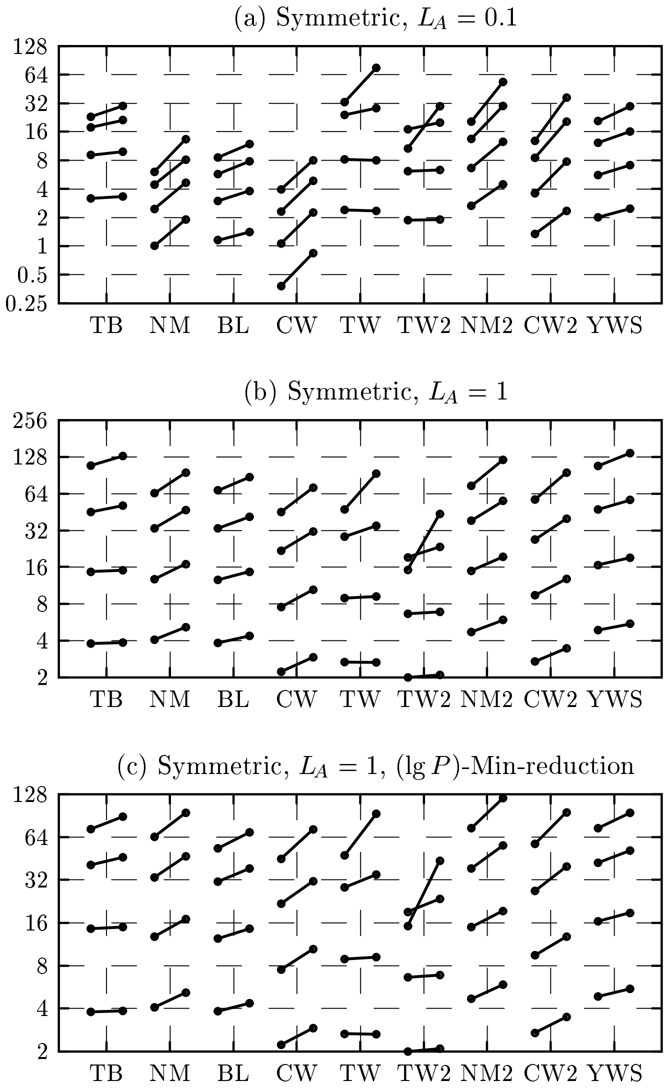


Figure 3: Predicted speedups for 4, 16, 64 and 256 processors. Segments represent speedups for each processor and protocol, with data for low load (segment's left point) and high load (segment's right point). The y-axis shows speedups and the x-axis identifies protocols.

nication such as TB and YAWNS achieve better speedups than the asynchronous protocols.

However, the speedups in figure 3.b do not consider the cost of performing periodical min-reductions in the associated synchronous protocols. Let us assume that we use a min-reduction operation which requires $\lg P - 1$ supersteps to complete the minimum calculation and its respective broadcast. The BSP cost of this method is $\approx [1+g+l] \lg P$ though with small constant factors for computation and communication. We only consider the cost of synchronization in this operation. Figure 3.c shows significant reduction of speedups in the synchronous protocols for $P = 256$ and 64 as a result of the additional supersteps introduced by the min-reductions.

Another fact observed in figure 3.b is that under large lookahead $L_A = 1$, the conservative protocols identified with NM and CW achieve reasonable performance. These protocols use less knowledge of the simulation model to increase lookahead than their counterparts NM2 and CW2. In most cases NM (and CW) outperforms TW.

Figure 4.a shows speedups for the *asymmetric* (favorite link) version of the queuing network. All speedups values decrease noticeably with this workload. However, TW remains comparatively efficient under this demanding case. Synchronous protocols achieve reasonable performance as well. For small number of processors TB outperforms TW.

The following two figures show the effects of increasing the values of the l and g parameters. These speedups should be compared with those shown in figure 3.a. Figure 4.b presents speedups for a case in which the synchronization cost l is 10 times larger than the benchmark values of the Cray T3D. The figure shows smaller speedup values than the figure 3.a. However, as expected, the greater reductions of speedups are observed in the synchronous protocols as they require more event-processing supersteps to complete the simulation of the workload. This penalty in performance is independent of the min-reduction realization. The asynchronous protocols (e.g., TW and NM2) get their speedups degraded in a smaller degree.

Asynchronous protocols have higher requirements of communication than synchronous ones since they must send anti-messages and null-messages. Figure 4.c shows the effects of increasing 100 times the cost of communication g. Now we see greater degradation of speedups in the asynchronous protocols. However, this reduction is not enough to make these protocols significantly less efficient than the synchronous ones.

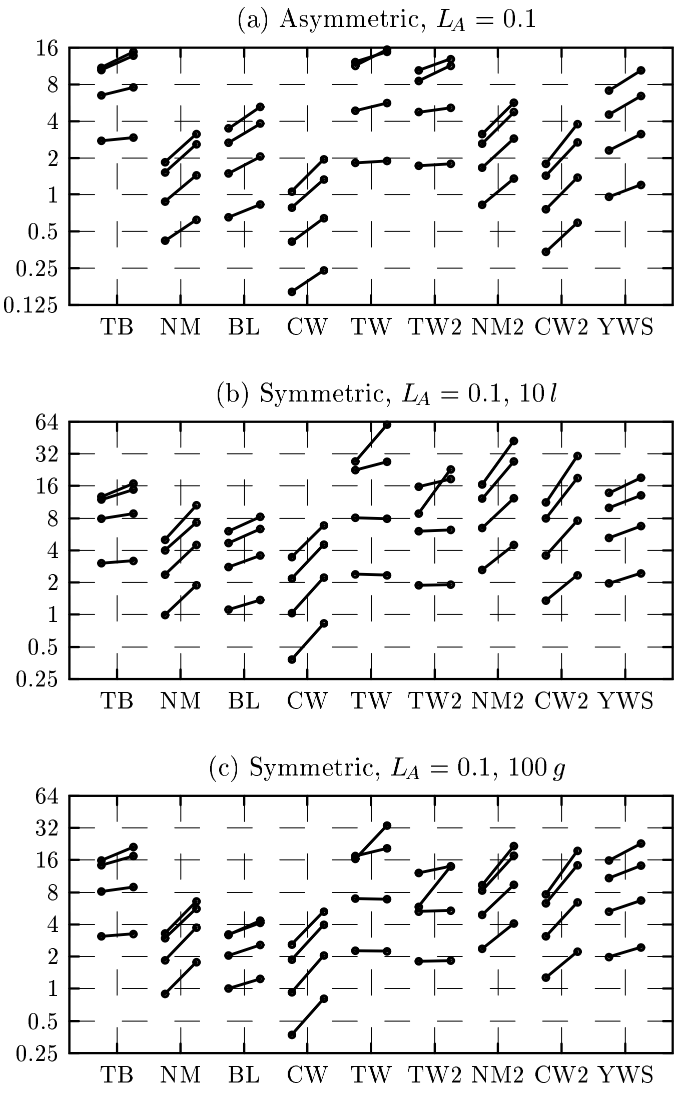


Figure 4: Predicted speedups (part 2) for 4, 16, 64 and 256 processors. Segments represent speedups for each processor and protocol, with data for low load (segment's left point) and high load (segment's right point). The y-axis shows speedups and the x-axis identifies protocols.

5 Conclusions

Our results confirm some well-known facts about conservative protocols Their performance improves noticeably with the amount of lookahead available in the model and the number of messages (load). The performance of the optimistic protocol is also improved by these factors. CMB can outperform TW in cases of large lookahead and high load. TW is able to remain efficient under small lookahead and low load.

We have also observed that the software overheads of TW are only slightly larger than the overheads of the conservative simulation. In addition, this protocol has the lowest requirements of synchronization which enables the simulation of more events per superstep. This improves the load balance. Also the anti-message traffic does not increase noticeably the cost of communication. The communication requirements of TW are lower than CMB with null-messages.

For a large lookahead, the synchronous and asynchronous *conservative* protocols achieve similar performance. In this case the total number of supersteps tends to be similar in each protocol. For a small lookahead, the asynchronous protocols outperform the synchronous protocols. In these cases, the synchronous protocols require larger number of supersteps to complete the simulation. As a result they are strongly impacted by a moderately high cost of barrier synchronization of processor. These protocols are more suitable for communication topologies with large fan-in/fan-out. The need for performing periodical min-reductions can also be detrimental to the performance of the synchronous protocols in systems with sparse communication topology.

Some features of the conservative synchronization protocols studied in this paper are the following:

- TB uses a conservative global time window. The protocol is simple and can be efficient in work-loads with very large lookahead. The protocol is independent of the communication topology among LPs.

- CMB-NM uses local time windows and achieves efficient performance in communication topologies with small fan-in/fan-out.

- CTW-SP has similar features to CMB-NM though the periodical execution of the shortest paths algorithm can increase overheads.

- BL requires periodical examination of the LPs located in a pre-defined neighborhood. This exacerbates its dependency of the communication topology and increases overheads.

- YAWNS is independent of the communication topology and can achieve efficient performance in work-loads with moderate lookahead.

Overall, for the conservative simulation protocols and under situations of good lookahead, asynchronous time advance protocols such as realized in CTW-SP or CMB-NM can be a more efficient alternative to the synchronous time advance imposed by protocols such as YAWNS. However, protocols such as CTW and CMB are restricted to communication topologies with small fan-in/fan-out. YAWNS does not consider the communication topology in its operation. An extreme case would be a fully connected topology where each LP may send messages to any other LP. In this case, adherence to the input rule in either CMB or CTW becomes a too expensive operation. We believe that any conservative protocol which bases its operation on local information (at LP level) needs to consider the communication topology in order to determine which events are safe to simulate in a given superstep. In particular, it is necessary to know the highest lower bounds for the timestamps of the next events to arrive from other LPs. YAWNS overrides this problem by calculating a global lower bound for the times of the next events in each LP. Nevertheless, the above results show that this can have a significant penalty in performance for practical systems in which the fan-in/fan-out of the communication is small.

As a validation of the previous analysis, figure 5 shows experimental results with actual implementations of the protocols. This is a favorable case for conservative synchronous protocols since the number of processors and logical processes is small and the available lookahead is large.

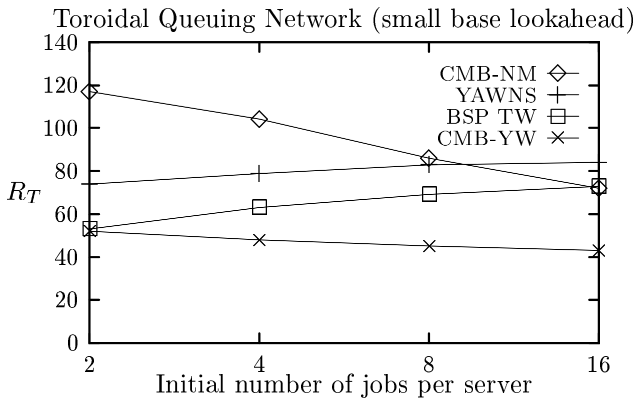
Acknowledgment

This work has been partially funded by research project Fondecyt 1990627.

References

[1] R. Ayani and H. Rajaei. "Parallel simulation using conservative windows". In *1992 Winter Simulation Conference*, pages 709–717, 1992.

[2] R.M. Fujimoto. "Parallel discrete event simulation". *Comm. ACM*, 33(10):30–53, Oct. 1990.



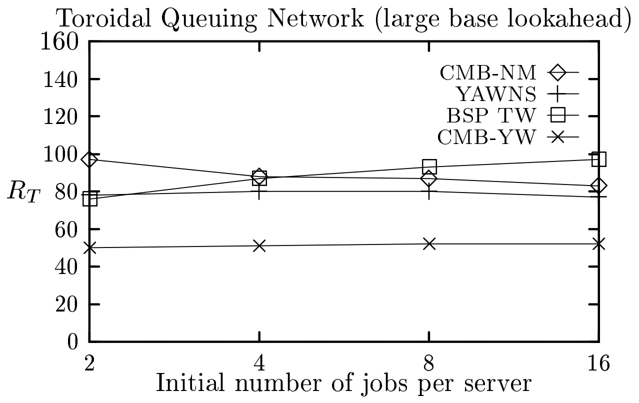


Figure 5:
Results for a BSPlib implementation of the protocols on a 4-processors SGI PowerChallenge computer. R_T= running time in seconds. Figure (a) shows values for a case in which the time of the next events are calculated by the sum $0.1 + X$ and figure (b) for the case $1.0 + X$ (the constants 0.1 and 1.0 — called base lookahead — are exploited by the conservative protocol to achieve better performance). In the figures, CMB-NM is CMB with null-messages, whereas CMB-YW is CMB-NM extended with YAWNS' lookahead.

[3] D.R. Jefferson. "Virtual Time". *ACM Trans. Prog. Lang. and Syst.*, 7(3):404–425, July 1985.

[4] B.D. Lubachevsky. "Efficient distributed event-driven simulations of multiple-loop networks". *Comm. ACM*, 32(1):111–123, Jan. 1989.

[5] M. Marín. "Parallel discrete-event simulation on the bulk-synchronous parallel model". PhD Thesis, Programming Research Group, Computing Laboratory, Oxford Unversity, 1998 (anonymous ftp at ona.fi.umag.cl).

[6] J. Misra. "Distributed discrete-event simulation". *Computing Surveys*, 18(1):39–65, March 1986.

[7] D.M. Nicol. "The cost of conservative synchronization in parallel discrete event simulations". *Journal of the ACM*, 40(2):304–333, April 1993.

[8] D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. "Questions and answers about BSP". Technical Report PRG-TR-15-96, Computing Laboratory, Oxford University, 1996. Also in *Journal of Scientific Programming*, V.6 N.3, 1997.

[9] J.S. Steinman. "SPEEDES: A multiple-synchronization environment for parallel discrete event simulation". *International Journal in Computer Simulation*, 2(3):251–286, 1992.

[10] L.G. Valiant. "A bridging model for parallel computation". *Comm. ACM*, 33:103–111, Aug. 1990.