Automatic Parallel-Discrete Event Simulation

Mauricio Marín[*]

Centro de Estudios del Cuaternario CEQUA
Universidad de Magallanes
Punta Arenas, Chile
Mauricio.Marin@umag.cl

Abstract. This paper describes a software enviroment devised to support parallel and sequential discrete-event simulation. It provides assistance to the user in issues such as selection of the synchronization protocol to be used in the execution of the simulation model. The software framework has been built upon the bulk-synchronous model of parallel computing. The well-defined structure of this model allowed us to predict the running time cost of synchronization protocols in accordance with the particular work-load generated by the simulation model. We exploit this feature to automatically generate the simulation program.

1 Introduction

Discrete-event simulation is a widely used technique for the study of dynamic systems which are too complex to be modelled realistically with analytical and numerical methods. Amenable systems are those that can be represented as a collection of state variables and whose values may change instantaneously upon the occurrence of events in the simulation time. It is not difficult to find real-life systems with associated simulation programs which are computationally intensive enough to consider parallel computing, namely *parallel discrete-event simulation* (PDES) [1], as the only feasible form of computation.

Over the last decade or so, parallel discrete event simulation (PDES) has been intensively performed on traditionals models of parallel computation [1,7]. Paradoxically, these models have failed to make parallel computation a paradigm of wide-spread use. Among other drawbacks, they lack realistic cost models for predicting the performance of programs. As a result, existing simulation software products have not been built upon frameworks which are able to predict performance and select accordingly the most efficient algorithms for implementing the parallel simulation program. Usually this kind of software either encapsulate just one fixed algorithm or leaves to the user the responsability of selecting one from a set of alternative ones. Neither is convenient since it is known that a particular algorithm is not efficient for all work-load cases and the target user not necessarily is an expert on efficient algorithms for parallel simulation. This has restrained potential users from using existing parallel simulation software.

[*] Partially supported by Fondecyt project 1030454.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3038, pp. 480–487, 2004.

In this paper we describe a simulation environment whose design is based on the use of the bulk-synchronous model of parallel computing (BSP model) [8, 11]. The key point is that the model of computing provides a simple way to cost parallel algorithms in their computation, communication and synchronization components. This allows us to predict the performance of different synchronization protocols in accordance with the particular features of the user defined simulation model. We use this facility to automatically select the synchronization protocol that is best suited for the particular simulation.

2 Object-Oriented Approach to Modeling

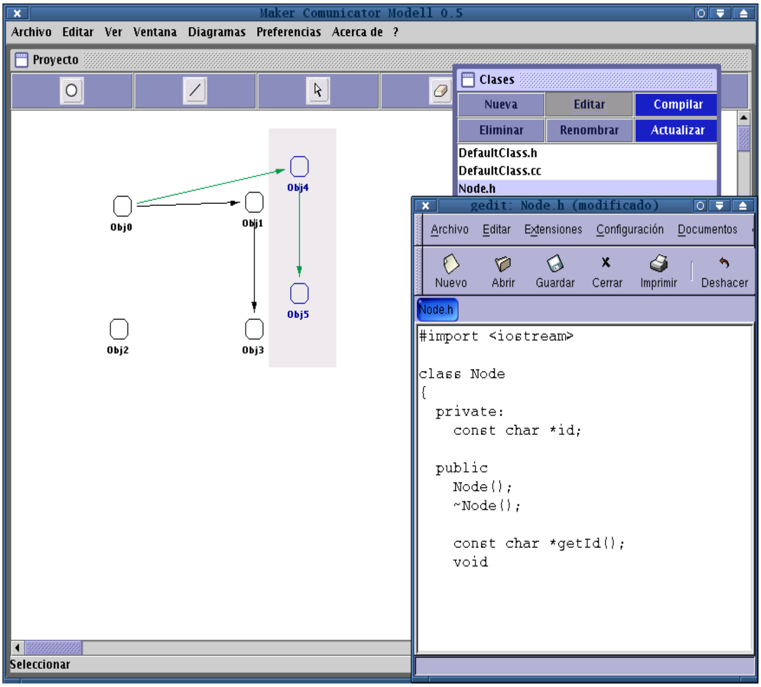
We describe a simple general purpose modeling methodology (the *world-view* in simulation parlance) that we have devised to facilitate the use of our system.

The world is seen as a collection of simulation objects that communicate with each other via timestamped event-messages. Associated with each object there is a global instance identifier, called object-id, and a class identifier, called entity-id. There exists a simulation kernel that is responsible for efficiently delivering the messages in strict message timestamps chronological order. Each simulation object inherits from a base class called *Device* that provides methods for the interface with the kernel. In particular, the kernel delivers messages to the simulation objects by executing the Device's method *cause* with parameters such as event-type, object-id, entity-id, and the simulation time at which the event takes place in the target object.

For each simulation object, the user must provide an implementation of the cause method so that events are handled in accordance with the behaviour defined for the particular object. Nevertheless, we enable users to maintain a class library for pre-defined/common use simulation objects together with a graphical representation for the user interface. Groups of classes can be tailored to specific application domains. The entity-id parammeter allows the user to split the event processing task into a set of private methods, each handling different types of events for a given type of entity.

In addition, simulation objects contain output channels that they use for sending messages to other objects connected to those channels. A message is sent out by executing the Device's method *schedule* which takes parammeters such as the channel id, time at which the event must take place in the target object, and sender's object-id and entity-id. At initialization time, all output channels are connected to their respective target objects. Note that the cause method could work on input channels as well. However, we have not seen a need for including in our world-view the notion of input channels yet. In fact, the combination object-id/entity-id appears to be very flexible as it allows objects to receive messages from multiple sources without complicating too much the initialization process.

On the other hand, the notion of output channels makes it easier to the programmer to work on generic implementations of the object without worrying about the specific object connected to the output channel. Actually, all this is



```
#import <iostream>

class Node
{
  private:
    const char *id;

  public
    Node();
    ~Node();

    const char *getId();
    void
```

Fig. 1.

a tradeoff between generality and simplification of the initialization process and its implicancies in code debuging and maintenance. As an alternative to output channels, we also support the concept of associative arrays (e.g., C++ maps) that are used by *schedule* to go from global object ids to pointers to the actual simulation objects for direct message delivering.

Currently, we have C++ and Java implementations of this approach.

The graphical user interface is designed to ease the burden of deploying thousands of simulation objects and defining their communication relations. The model definition process starts up with the creation of a project and drawing of a few objects. Each object is an instance of a given class which can be edited and/or viewed by using a class editor as shown in figure 1. More objects of a given class can be deployed by duplicating existing objects. The user interface allows the definition and/or edition of different properties associated with the objects as well as the automatic generation of the simulation program.

The architecture of the simulation framework is as follows. On the upper layer we have the user interface which allows the simulation model to be defined in a graphical manner as well as the codification of the classes that define the behavior of the respective simulation objects. The second layer takes the user definitions and generates a specification of the model written in a mid-level language. The third layer is in charge of selecting the synchronization protocol that happears to be most suitable for the simulation model. This is effected by directly

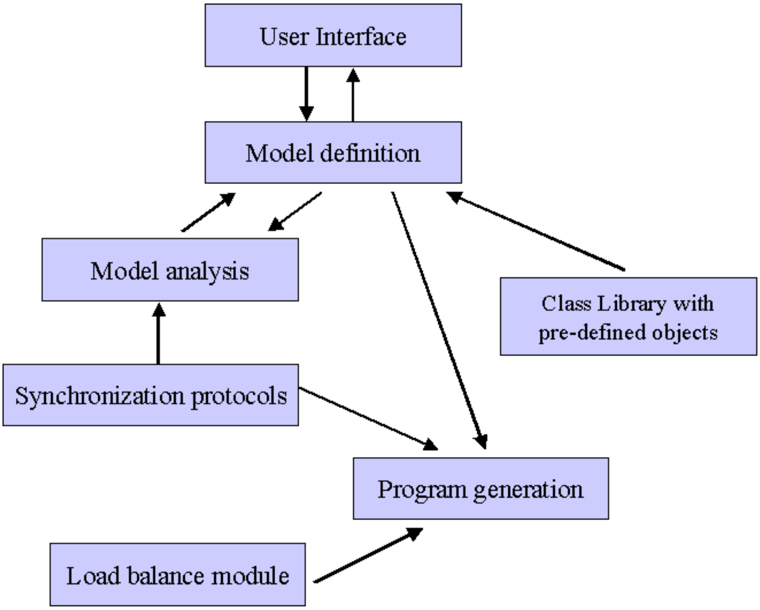


Fig. 2. Software Architecture

analyzing the mid-level definitions. The first step is to perform a pre-simulation of the model. The result is a set of parameters that predicts the model behavior. These parameters are then input to a formula that predicts whether it is better to simulate the model in parallel or just sequentially in the target parallel computer. The last step is to generate the simulation program by linking the selected synchronization protocol with the simulation objects defined by the user. Finally a C++ or java compiler is used to produce the executable simulation program.

Figure 2 shows the relationships among the main components of the simulation environment. Note that users who are not familiar with C++/Java programming can profit from the class library that contains definitions for most typical objects in, for example, queuing systems. The library can be increased by directly including new user-defined class definitions.

Usually a parallel simulation will have to deal with thousands of simulation objects. Thus the mid-level languaje only specifies general information about the objects such as the class they belong to and their communication relations. The instances themselves are stored in a symbolic manner into a file to be actually created later at simulation running time.

The methodology used to automatically generate the simulation program can be divided into the following major steps (see figure 2):

(**i**) Pre-simulation of the simulation model defined by the user. This uses information of the communication topology among objects defined by the user and the details about what random number generators are used to send the messages among them (simulation time of those event messages). The results of the pre-simulation are used to determine a tuple which describes the overall behaviour of the model in terms of the amount of computation, communication and synchronization it demands to the BSP computer per unit simulation time (see next section).

(**ii**) The tuple obtained in the previous step is plugged into a formula that predicts the feasible speedup to be achieved by the simulation on the target BSP computer (next section). The outcome can be a recomendation to simulate the model just sequentially because the predicted speedup is too modest or even less than one. The effect of the particular parallel computer hardware is included in the BSP parameters G and L obtained for the machine via proper benchmarks (next section).

(**iii**) In the case that the recomendation is a parallel simulation, the tuple is now plugged into a set of formulas that predict the running times of a set of synchronization protocols available in our system for parallel simulation. These protocols are optimistic and conservative ones and for each of them it is necessary to include new definitions into the simulation model. Conservative protocols need the so-called lookhaead information whereas the optimistic one requires the specification of what are the states variables that need to be handled by rollbacks.

(**iv**) The simulation program is generated by putting together objects and synchronization protocol. During simulation the synchronization protocols have the avility of adapting themselves to changes in the work-load evolution. Those protocols also implement a dynamic load balancing strategy we devised to re-distribute objects onto processors in order to reduce running times (figure 2). The initial mapping of objects onto the processors is uniformly at random. During running time a dynamic load balancing algorithm is executed to correct observed imbalance.

3 Technical Details

In the BSP model of computing both computation and communication take place in bulk before the next point of global synchronization of processors. Any parallel computer is seen as composed of a set of P processor-local-memory components which communicate with each other through messages. The computation is organised as a sequence of *supersteps*. In each superstep, the processors may perform sequential computations on local data and/or send messages to other processors. The messages are available for processing at their destinations by the next superstep, and each superstep is ended with the barrier synchronisation of the processors.

The total running time cost of a BSP program is the cumulative sum of the costs of its supersteps, and the cost of each superstep is the sum of three quantities: w, $h\,g$ and l, where (**i**) w is the maximum of the computations performed by each processor, (**ii**) h is the maximum number of words transmitted in messages sent/received by each processor with each one-word-transmission costing g units of running time, and (**iii**) l is the cost of barrier synchronising the processors.

The effect of the computer architecture is costed by the parameters g and l, which are increasing functions of P. These values can be empirically determined by executing benchmark programs on the target machine [8].

We use the above method to cost BSP computations to compute the speedup S_{up} under a demanding case for the underlying parallel algorithm. In [4] we derived the following speed-up expression,

$$S_{up} = \frac{1}{P_B\,(1 + P_M/r)\ +\ z\,P_B\,P_M\,g_e\ +\ P_S\,l_e}$$

with $\frac{1}{P} \leq P_B \leq 1$, $0 \leq P_M \leq 1$, $0 \leq P_S \leq 1$, $r \geq 1$, and $z \geq 1$. The parameter P_S is a measure of slackness since $1/P_S =$ number of simulated events per superstep. The parameter P_M accounts for locality as it is the average fraction of simulated events that results in message transmitions. The parameter P_B accounts for load balance of the event processing task. The size of messages is represented by z. In addition, $r \geq 1$ is the event granularity defined with respect to C_e which is the lowest (feasible) cost of processing an event in the target machine. Finally, g_e and l_e are defined as $g_e = g/(r\,C_e)$ and $l_e = l/(r\,C_e)$ for the BSP parameters g and l respectively. In this way simulation models can be represented by an instance of the tuple (P_B, P_S, P_M, r, z).

The most popular synchronisation protocols [2,5,6,10] base their operation on one of two strategies of simulation time advance. Synchronous time advance (SYNC) protocols define a global time window to determine the events allowed to take place in each iteration (superstep). The SYNC protocol advances its time window forward in each iteration to let more events be processed. On the other hand, asynchronous time advance (ASYNC) protocols implicitly define windows which are local to the simulation objects. Figure 3 describes sequential simulation algoritms which predict the supersteps executed by each protocol (SYNC and ASYNC).

The comparative cost of synchronization protocols is calculated as follows. Let S_p^s and S_p^a be the number of supersteps per unit simulation time required by synchronous time advance (SYNC) or asynchronous time advance (ASYNC) respectively. We measure load balance (at superstep level) in terms of the *event efficiency* E_f as follows. If a total of $M_e \gg N$ events are processed during the complete simulation with P processors, then $E_f = \frac{M_e}{\text{SumMaxEv}}\,\frac{1}{P}$ where *SumMaxEv* is the sum over all supersteps of the maximum number of events processed by any processor during each superstep (i.e., the cumulative sum of the maximum in each superstep). Both S_p and E_f can be determined empirically for the simulation model at hand by employing the algorithms shown in figure 3.

The protocols in our system are optimized BSP realizations of YAWNS [6], BTB [9], CMB-NM (null messages) [5] and TW [2]. YAWNS and BTB are SYNC protocols whereas CMB-NM and TW are ASYNC protocols.

In the optimistic protocols we increase the cost of processing each event in $\varphi \geq 1$ units in order to include the effect of state saving. Roll-backs cause re-simulation of events thus we consider that this operation increases the total

```
SYNC
  Generate $N$ initial pending events;
  [ $e$ is an event with time $e.t$ ]
  $T_Z := \infty$; [ event horizon time ]
  $S_Z \leftarrow \Phi$; [ buffer ]
  loop
      if TimeNextEvent() $> T_Z$ then
          SStep := SStep + 1;
          Schedule($S_Z$);
          $T_Z := \infty$;
          $S_Z \leftarrow \Phi$;
      endif
      $e :=$ NextEvent();
      $e.t := e.t$ + TimeIncrement();
      $p := e.p$; [ $e$ occurs in processor $p$ ]
      $e.p :=$ SelectProcessor();
      if $e.p \neq p$ then
          $S_Z \leftarrow S_Z \cup \{e\}$;
          $T_Z :=$ MinTime($S_Z$);
      else
          Schedule($e$);
      endif
  endloop
```

```
ASYNC
  Generate $N$ initial pending events;
  [$e.s$ indicates the minimal superstep at
    which the event $e$ may take place in
    processor $e.p$.]
  loop
      $e :=$ NextEvent();
      $p := e.p$; [ $e$ occurs in processor $p$ ]
      if $e.s >$ SStep[$p$] then
          SStep[$p$] := $e.s$;
      endif
      $e.t := e.t$ + TimeIncrement();
      $e.p :=$ SelectProcessor();
      if $p = e.p$ then
          $e.s :=$ SStep[$p$];
      else
          $e.s :=$ SStep[$p$] + 1;
      endif
      Schedule($e$);
  endloop

  The total number of supersteps is the
  maximum of the $P$ values in array SStep.
```

Fig. 3.

number of simulated events by a factor of ϕ events with $\phi \geq 1$. In the asynchronous protocol roll-backs also increase the message traffic. The conservative protocols do not have these overheads thus we set $\varphi = 1$ and $\phi = 1$. Synchronous time advance protocols (SYNC) require a min-reduction operation with cost R_D for each event processing superstep.

Defining N and N_m as the number of simulated events and sent messages per unit simulation respectively on a P-processors BSP computer, the total cost of a SYNC protocol (YAWNS and BTB) is given by SYNC $= \frac{\varphi^s \, \phi^s \, r \, N}{E_f^s \, P} + \frac{N_m}{E_f^s \, P} \, g + S_p^s \, l + S_p^s \, R_D$, whereas the total BSP cost of the ASYNC protocol (TW and CMB-NM) is given by ASYNC $= \frac{\varphi^a \, \phi^a \, r \, N}{E_f^a \, P} + \frac{(2\,\phi^a - 1)\, N_m}{E_f^a \, P} \, g + c \, S_p^a \, l$. where $c \geq 1$ is a factor that signal the average increase of supersteps in CMB-NM. The determination of the synchronization protocol to be suggested to the user takes into account the following cases. Conservative protocols (YAWNS, CMB-NM) have higher priority than the optimistic ones (BTB, TW). For the case in which the observed fan-in/fan-out of the communication topology among the simulation objects is large enough, the CMB-NM is discarded since this protocol loses efficiency dramatically. Also for the cases in which the pre-simulation did not find a sufficient amount of "lookahead" in the built-in random number generators for timestamps increments, the YAWNS and CMB-NM protocols are discarded. On the other hand, the cost of state saving for the optimistic protocols depends on the size of data to be periodically saved. Also roll-backs in BTB are on everage 20% lower than in TW. Thus the best choise is a tradeoff.

4 Conclusions

We have described the overall design of a simulation framework we have developed to support parallel discrete event simulation. The main objective was to assist users on the complexity associated with the selection of a proper synchronization protocol to conduct the simulation of the user-defined model.

A pre-simulation of the user model produces information about what synchronization protocol is best suited for the execution of the model. Those protocols are devised upon a model of computing that provides both independence of the architecture of the parallel computer and a method for determining the cost of parallel algorithms.

We have tested the suitability of our system using several simulations models. Those include the synthetic work-load PHold, Wind energy electricity generation systems, hard-particles models, Web crawlers, and a large toriodal queuing network. In all cases, specially in regular systems, we have observed that our prediction methodology is very effective in practice (this claim is supported by the results in [3,4]).

References

1. R.M. Fujimoto. Parallel discrete event simulation. *Comm. ACM*, 33(10):30–53, Oct. 1990.
2. D.R. Jefferson. Virtual time. *ACM Trans. Prog. Lang. and Syst.*, 7(3):404–425, July 1985.
3. M. Marín. Asynchronous (time-warp) versus synchronous (event-horizon) simulation time advance in bsp. In *Euro-Par'98 (Workshop on Theory and Algorithms for Parallel Computation*, pages 897–905, Sept. 1998. LNCS 1470.
4. M. Marín. Towards automated performance prediction in bulk-synchronous parallel discrete-event simulation. In *XIX International Conference of the Chilean Computer Science Society*, pages 112–118. (IEEE-CS Press), Nov. 1999.
5. J. Misra. Distributed discrete-event simulation. *Computing Surveys*, 18(1):39–65, March 1986.
6. D.M. Nicol. The cost of conservative synchronization in parallel discrete event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.
7. D.M. Nicol and R. Fujimoto. Parallel simulation today. *Annals of Operations Research*, 53:249–285, 1994.
8. D.B. Skillicorn, J.M.D. Hill, and W.F. McColl. Questions and answers about BSP. *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. J.S. Steinman. Discrete-event simulation and the event-horizon. In *8th Workshop on Parallel and Distributed Simulation (PADS'94)*, pages 39–49, 1994.
11. L.G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33:103–111, Aug. 1990.