Two-Level Scheduling Algorithm for Simulation Servers

Mauricio Marín Carolina Bonacic
Departamento de Computación
Universidad de Magallanes
Casilla 113-D, Punta Arenas, Chile
E-mail: {cbonacic,mmarin}@ona.fi.umag.cl

Abstract

We present a scheme for efficiently administrating a set of distributed simulation servers which attend requests from a number of users. These can be either sequential or parallel simulation models, or random combinations of the two. We treat each server as a BSP machine and organize our scheduling strategy upon the cost prediction and computation model supported by it. Empirical results show that the proposed strategy is able to achieve near optimal performance. Our aim was to develop a mechanism which be able to work with minimum global information so that it can be utilized on Web based simulation systems.

1 Introduction

The Web has became an obiquitous resource for distributed computing making it relevant to investigate new ways of providing efficient access to services available at dedicated servers from Web pages. This paper is concerned with the efficient administration of a set of discrete-event simulation servers which are able to execute user's simulation models in parallel or sequentially.

The scheme proposed consists of a number of instances of three basic entities which we call *models*, *managers* and *servers*. The first one represents user simulation models that are sent to managers, which in turn decide to which simulators (servers) the models are scheduled for execution. The metric to optimize is *response time* defined as the total real time elapsed between model submission and simulation results delivery.

Parallel simulations are carried out using bulk-synchronous parallel (BSP) processing [11] which not only allows the achievement of high performance computation but also the accurate prediction of that performance in advance, which is crucial to our scheduling purposes. At every server site, what we actually have is a set of P processors (e.g., a cluster of PCs) which are treated either as a single machine for the case of executing a parallel model requiring P processors, or as a set of P individual machines capable of executing up to P sequential simulation models during a given period of time. Communication and synchronization among these P processors is performed by ways of and paradigm supported by the BSP model of computing [12, 11]. Communication among the many instances of model, manager and server entities is supported by the CORBA technology.

The paper describes novel developments. First, it describes how to use (in this context) a cost prediction methodology we have developed for predicting performance of parallel and sequential simulations. This methodology provides the basic framework upon which the scheduling process is carried on. Second, it proposes a way of representing simulation models, and in general a unifying way of dealing with their complexity as a whole, which simplifies the scheduling process. Third, it proposes a method for administrating the execution of parallel and sequential simulations models which is based on the organization of the available computers as a cluster synchronized using the BSP model of computing. This allows us to dynamically allocate a number of computers (processors) to either simulate a sequential model each or a single parallel model. The proposed framework allows managers to decide whether or not it is worth to execute in parallel a given simulation model on the target machine. To the best of our knowledge the combination of these topics has not been investigated so far, though the parallel simulation literature is extensive and diverse (e.g., see surveys in [1, 8, 9, 10, 4]). None of this has been investigated in the BSP context, in particular, the feasibility of using a BSP cluster of PCs to dynamically service simulation workloads in parallel and sequentially.

2 Running times

The running time of parallel discrete event simulations is predicted in terms of the bulk-synchronous parallel (BSP) model of computing [12, 11]. In BSP, 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*. During a 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 w is the maximum of the computations performed by each processor, h is the maximum of the messages sent/received by each processor with each word costing g units of running time, and l is the cost of barrier synchronising the processors. The effect of the computer architecture is cost by the parameters g and l, which are increasing functions of P. These values along with the processors' speed s (e.g. mflops) can be empirically determined for each parallel computer by executing benchmark programs at installation time [11].

In [7, 6, 4] we have presented formulae for predicting the running times of different synchronization protocols for performing parallel discrete-event simulation on BSP computers. In particular, we have obtained an expression for determining whether or not it is convenient to resort to parallel simulation for a given model,

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

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, namely it is the inverse of the average amount of events processed in each superstep. The parameter P_M accounts for locality, and it determines the fraction of communication that every simulated event generates. The parameter P_B accounts for load balance where the optimum is reached at $P_B = 1/P$ with P being the number of processors. The granularity of events is captured by $r \geq 1$ which is a factor that increases the lowest (feasible) cost C_e of processing an event in a particular machine. Similar to the BSP parameters g and l the value of C_e is determined by performing benchmarks on the target computer. We also define $g_e = g/(r\,C_e)$ and $l_e = l/(r\,C_e)$. Finally, z is the size of event messages transmited among the simulation objects. The effectiveness of this strategy has been validated with empirical data from actual simulators [6, 4].

In this way simulation models can be represented by an instance of the tuple $(P_B, P_S, P_M, r, z, T_r)$ where T_r is an estimation of total running time required by the simulation (below we explain how to calculate this measure). The expression for the speedup S_{up} is useful in the sense that it can let our *manager* entities determine whether or not it is worth to route a given simulation model to a given parallel simulation *server*. Models not only should be routed to the servers which offer the best possible speedups but also to those which are not busy enough to make response time larger than that achieved by a more modest sever.

Similar arguments to those used in the determination of S_{up} can be used to predict running times per unit simulation time of simulation models to be executed sequentially or in parallel. The managers can use these expressions to make decisions about which servers should be allocated to what models.

As explained in section3, we use information about the communication topology among simulation objects to determine the tuples $(P_B, P_S, P_M, r, z, T_r)$. The objects by themselves provide information about the number of events to be executed during the simulation. As it is not generally possible to know the exact sequence of events that take place during the simulation but until they are actually simulated, we assume that these objects behave like PHold entities [2, 3, 4]. That is, initially each object causes the occurrence of one event, and thereafter each event that takes place generates the occurrence of a child event in a neighboring object selected uniformly at random. The interval of simulation time elapsed between father and child events is, for example, exponentially distributed with mean one. This certainly provides just an approximation of the real event process. However, all simulation models are treated the same way which provides a common cost modelling framework. The outcome of this pre-simulation is an estimation for T_r.

Thus the *model* entities send tuples $(P_B, P_S, P_M, r, z, T_r)$ to their *manager* entities along with an estimation of the total execution time required to complete the simulation. In turn the *managers* compute running time expressions and determine which *server* entities should the models be executed. This is effected by considering the current models being executed at servers and the pending models waiting for execution. This is detailed in section 4.

3 Worldview

A number of the above calculations depend on the way of constructing the simulation models. They are built up by using a specific methodology which presents to the user a particular *world-view*. The one which we have devised allows the automatic determination of the tuples (P_B, P_S, P_M, r, z) by means of performing a pre-simulation of the models (one which only considers the communication topology among simulation objects). This in combination with benchmarks oriented to determine lower bounds for event execution times [6] provides the necessary data for the scheduling decisions.

In the world-view, any system is seen as a composed of a collection of 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 chronological order given by the message timestamps. Each simulation object inherits from a base class called *Device* that provides methods for interfacing the kernel. In particular, the kernel delivers messages to the simulation object 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. The entity-id parameter 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, simulations 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 channel number, time in 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 a tradeoff between generality and simplification of the initialization process and its implicancies in code debuging and maintenance. Currently, we have C++ and Java implementations of this approach and its effectivity has been proven with undergraduate level students' simulation projects.

The Device class is instrumented to allow a rapid pre-simulation (PHold) of the simulation objects so that the associated model tuple and total execution time are estimated. Currently, these pre-simulations are performed at the user site though they could alternatively be effected at the manager site. Parallel simulations are performed by using the event synchronization protocol described in [5].

4 Scheduling strategy

Users can submit sequential or parallel simulation models written in a especial purpose specification language (a text file). First, a tuple $(P_B, P_S, P_M, r, z, T_r)$ describing such model is sent to the manager associated with the particular instance of the user entity. This tuple can be determined from the specification file either at the user's machine or at the manager's machine. The file describes the communication topology and the random number generators (mean values included) all of which is used by the pre-simulator to estimate all parameters of the model tuple.

Each manager has the responsibility of administrating one server which is seen as a BSP machine composed of P processors (e.g., a cluster of PCs administered with the BSPlib communication library [11]). The scheduling process is effected in two levels as described in the following subsections.

4.1 Local scheduling

This level is related to individual manager-server pairs. The BSP machine (server) performs the simulation of either up to P different sequential models or a single parallel model. The server performs its computations using the above defined *superteps.* During a given supertep, we can have (up-to) P sequential models being simulated until their respective ends, or this superstep being one of many supersteps associated with the parallel simulation of a model requiring P or less processors. Instances of the user entities send their sequential simulation objects to one of the P machines, whereas the parallel simulation objects are sent to their respective processors as mapped by the programmer. This is effected after the manager has determined a particular scheduling for a set of users requesting service. We propose a simple but efficient scheme for this task.

Under a situation of intensive traffic we can expect to have a queue of user requests in a given manager. As we have a method for estimating the total running time of sequential and parallel simulation models, then the best thing to do comes from any operating systems textbook, namely, employing the optimal scheme called *the shortest job first*. The manager has two queues. The first one maintains the incoming user requests, say *jobs*, in FIFO order. The second one is feed periodically with batches of n jobs with $n/P > 2$ if possible. Thus a given batch is sorted in accordance with the estimated running times, and then the batch is re-ordered again trying to leave P sequential jobs between two parallel jobs. The batch is then sent to the server to be processed in alternate runs of P sequential models (one superstep) and one parallel model (several supersteps). In practice, we should expect a performance ranging between the above best case given by a batch composed by sequential models only, and the worst case given by a batch composed of parallel models only.

4.2 Global scheduling

The set of managers keep record of the number and type of user requests they locally receive. Initially, all BSP parameters (s, g, l, P) of simulation servers are known to all managers. During operation, periodically managers engage in a global all-to-all communication to let each other know the number of sequential models and parallel models they have received locally. Let N_s and N_p be the total number of sequential and parallel models received in the whole system respectively.

The aim of this global communication is to emulate the same idea of the local scheduling strategy. That is, for a global system composed of M managers/servers we want to approximate the operation of the whole system to the one of a Hiper-BSP machine with $M\,P$ processors having M batches to deal with. This for a situation in which all M BSP machines are identical. The M different values (s, g, l, P) are used to normalize things to this ideal scenario. Note that under this case it is sufficient to let each manager router every model it receives to another manager uniformly selected at random.

The cost involved in the global communication is assumed to be much smaller than the running times of simulation models (eventually some production simulation models are expected to be in the order of hours). It is performed on a regular basis to follow changes in the workload generated by the users. Frequency will be determined by the cost of the global communication. We choose this cost to be 1/10-th of the time elapsed between two consecutive communications.

The result after every communication is the determination of new routing probabilities for sequential and parallel models at each manager site. Thus a simulation model sent to a particular manager can be routed to other manager's FIFO queue with a certain probability calculated as follows. A job is routed to manager k with probability R_k/R_T where R_T is the sum of all R_i, and every R_i is calculated as

$$R_i = \frac{1}{N_s + N_p} \,(\, N_s\, P_i\, s_i \;+\; N_p\, [s_i + g_i + l_i]\,)$$

Note that this is different to the naive strategy of routing jobs to the managers with the smallest queues. Such strategy does not consider cases with imbalance in the number of parallel models and sequential models, and the relative differences between the BSP machines. The BSP parameters (s, g, l, P) correct differences among the machines so that a slower server site receives a smaller number of models. Also note that a model requiring more processors than the actual ones of the machine can be mapped in BSP using the approach of more than process per processor. On the other hand, parallel models can become sequential models if the predicted speedup S_{up} in their target servers is less than one.

5 Empirical evaluation

A sample system was built for the experiments presented in this section. It is Web crawling model composed by one scheduler and B robots running on a single host computer. Their mission is to gather documents stored on W web sites, each residing on a different host and containing a collection of documents. The Web is a graph with nodes split evenly on the W sites. Documents lengths and number of links per document are random variables. The H hosts are connected by a packet switched network consisting of E routers that use two phase shortest path routing. That is, first a given packet is sent to a randomly selected neighboring router, and then it is sent to its destination by following the shortest path. Links between pairs of routers are bi-directionals and have a certain transmition rate in bits per seconds.

Relevant computer processing time arises in three cases. First, when the scheduler stores a set of urls provided by one robot and then it retrieves the following url from the list. Second, when a robot receives a new document and processes it to extract url pointers to new documents. Third, Web site hosts requires some computer processing time when they retrieve a document and split it into a set of packages. Routers are modelled as FCFS queuing facilities with a number of servers equal to the number of communication links with other routers. Processing time is provided by objects of class *computer* implementing FCFS queuing with round-robin discipline. This for considering the effect of processing requests from multiple processes on a given host.

We first evaluated the effectiveness of our total running time estimation strategy. The results are shown in the column 2 of table 1. They show the ratio [estimated running time / actual running time] for 10 instances of the Web crawling model. In all instances we kept fixed $B = 50$, $H = 100$, $W/B = 10000$, and varied E from 1000 to 10000. The pre-simulations returned a particular event

Instance	T_r/real	dist/par	ratio 1	ratio 2
1	0.45	0.81	17.25	16.74
2	0.41	0.89	11.36	11.26
3	0.46	0.83	10.13	9.99
4	0.52	0.84	9.54	9.46
5	0.48	0.87	9.19	9.11
6	0.46	0.88	8.97	8.91
7	0.53	0.86	8.83	8.77
8	0.53	0.89	8.71	8.67
9	0.56	0.91	8.63	8.59
10	0.58	0.90	8.57	8.53

Table 1: Empirical results.

trace, and the running time cost of each event was estimated as follows: After inserting m events in the event-list, we performed a sequence of cm hold operations with $c \gg 1$. Each hold operation (**i**) retrieves from the event-list the event e with the least simulation time $e.t$, (**ii**) increases this time by $e.t + X$ units where X is a random variable with the distribution and mean value extracted from the specification file, and (**iii**) then re-insert this event e into the event-list. An accurate value of the cost of processing each event e can be obtained by measuring the total running time spent in processing the whole sequence of hold operations and dividing it by cm. The results show that the pre-simulation predicted fairly with the same error the total running times in all instances of the model, which is convenient as we are actually interested in the relative running times.

We employed simulation to evaluate the global scheduling strategy of the proposed scheduling strategy. We defined (simulated) a distributed system composed of 10 simulation servers with 8 processors each, and 500 users. We uniformly at random generated servers which are two and three times faster than a base server. In addition, for comparison purposes, we defined an average hiper-server with performance twice faster than the base server and with 80 processors attending itself the requests of the 500 users. In the column 3 of table 1 we show the ratio [performance distributed system / performance parallel system] for 10 instances of user behaviors and models, where performance is measured as the average response time experienced by the users. The results show that the scheduling strategy applied by the distributed system approaches well the parallel system.

The effectiveness of the local scheduling strategy was evaluated as follows. We implemented a simulation model capable of applying three methods of servicing a queue of jobs. They were FIFO, Round-Robin (RR) and shortest job first (SJF). The 10 instances of the experiments are given for the length of the queue, which ranged from 10 to 200 jobs. The ratio FIFO/SJF approx 5 and RR/SJF approx 3 was fairly constant for different combinations of the number of parallel and sequential jobs. Similar results were obtained for cases with large (1-100) and small (1-10) range of running times. Evidence that batches of size twice or three times P are more convenient than larger ones is in column 4 of the table. There it is shown the ratio [waiting time one processor / waiting time P processors]. It is observed that in small size queues it is possible to achieve the smallest time spent queuing. Column 5 shows the same but for a small range for random running times.

6 Conclusions

We have presented a scheduling strategy for the efficient administration of a set of simulation servers attending a number of users distributed on (possibly) the Internet. Users only need to have a pointer to one of a number of managers which are in charge of allocating servers to the users.

The proposed scheduling scheme is simple, requires little global information and, as the empirical results show, it is near optimal. A crucial aspect is that its effectiveness depends on the ability to predict the comparative performance of models on differing machines and types of models. This is possible due to the adoption of the BSP model of parallel computing which provides an architecture independent way of costing programs. We have developed sequential and parallel simulation models cost expressions which allows one to predict such comparative performance. By applying a pre-simulation from data collected from a text file with model specifications it is possible to estimate the parameters required by these expressions.

We currently have a prototypic implementation of the system which is composed of two cluster with 4 machines each, both residing in the same local network. We plan to extend this system in the near future.

References

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

[2] R.M. Fujimoto. "Performance of Time Warp under synthetic work-loads". In *SCS Multiconference on Distributed Simulation V.22 N.1*, pages 23–28, Jan. 1990.

[3] J. Liu, D. Nicol, B. Premore, and A. Poplawski. "Performance prediction of a parallel simulator". In *13th Workshop on Parallel and Distributed Simulation (PADS'99)*, 1999.

[4] M. Marín. 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).

[5] M. Marín. Time Warp on BSP Computers. In *12th European Simulation Multiconference*, June 1998.

[6] 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.

[7] M. Marín. Analysis of efficient synchronization in bulk-synchronous parallel discrete-event simulation. In *IASTED International Conference on Applied Simulation and Modelling.* (ACTA Press), June 2002.

[8] D.M. Nicol and R. Fujimoto. "Parallel simulation today". *Annals of Operations Research*, 53:249–285, 1994.

[9] D.M. Nicol, M.M. Johnson, A.S. Yoshimura, and M.E. Goldsby. "Performance modeling of the IDES Framework". In *11th Workshop on Parallel and Distributed Simulation (PADS'97)*, pages 38–45, 1997.

[10] D.M. Nicol, M.M. Johnson, A.S. Yoshimura, and M.E. Goldsby. "IDES: A Java-based distributed simulation engine". In *International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, Feb. 1998.

[11] 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.

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