Proceedings of the 1995 Winter Simulation Conference
ed. C. Alexopoulos, K. Kang, W. R. Lilegdon, and D. Goldsman

AUTOMATED LOAD BALANCING IN SPEEDES

Linda F. Wilson

Institute for Computer Applications
in Science and Engineering
Mail Stop 132C, NASA Langley Research Center
Hampton, Virginia 23681-0001, U.S.A.

David M. Nicol

Department of Computer Science
The College of William and Mary
P. O. Box 8795
Williamsburg, Virginia 23187-8795, U.S.A.

ABSTRACT

Parallel discrete-event simulation offers the potential for significant speedup over sequential simulation. Unfortunately, high performance is often achieved only after rigorous fine-tuning is used to obtain an efficient mapping of tasks to processors. In practice, good performance with minimal effort is often preferable to high performance with excessive effort.

In this paper, we discuss our research in adding automated load balancing to the SPEEDES simulation framework. Using simulation models of queuing networks and the National Airspace System, we demonstrate that using run-time measurements, our automated load-balancing scheme can achieve better performance than simple allocation methods that do not use run-time measurements, particularly when large numbers of processors are used.

1 INTRODUCTION

Ideally, parallel discrete-event simulation (PDES) is used to obtain significant speedups over sequential simulations. In reality, high performance is difficult to obtain. As noted by Nicol and Heidelberger (1995), "PDES is an unusually tricky branch of parallel processing" because high performance cannot be achieved unless the system is fine-tuned to balance computation, communication, and synchronization requirements. In particular, PDES cannot make a significant impact on practical discrete-event simulation unless tools are developed to automate the tuning process with little or no modification to the user's simulation code.

In a typical PDES, components of the system under examination are mapped into logical processes (LPs) that can execute in parallel. The LPs are distributed among the physical processors, and communication between LPs is accomplished by passing messages. If LPs are distributed among the processors such that interprocessor communication is minimized, some processors may sit idly waiting for something to do while others are overloaded with work. At the other extreme, a "perfectly-balanced" workload may yield poor performance due to high communication costs. Thus, load-balancing strategies must find a compromise between distributing work evenly and minimizing communication costs.

In this paper, we describe our early experiences in developing an automated load-balancing strategy for the SPEEDES simulation environment. Ultimately our goal is to use run-time measurements of simulation workload to guide automated remapping mechanisms. Towards that end we investigate here the increased performance one might achieve using a load-balancing algorithm that uses run-time measurements, as opposed to those that do not. Section 2 presents background material on SPEEDES while Section 3 describes the load-balancing methodology and the modifications made to SPEEDES to automate the process. Section 4 presents results from two models simulated on the Intel Paragon and compares the execution times obtained by load balancing, default partitioning, and card dealing. Section 5 presents our conclusions.

2 SPEEDES

SPEEDES (Synchronous Parallel Environment for Emulation and Discrete-Event Simulation) is an object-oriented simulation environment that was developed at the Jet Propulsion Laboratory (Steinman 1991, 1992). Designed for distributed simulation, SPEEDES supports multiple synchronization strategies (including Time Warp, Breathing Time Buckets, and Breathing Time Warp) that can be selected by the user at runtime. In addition, SPEEDES provides a sequential simulation mode (with most of the parallel overhead removed) so that a particular simulation model can be executed serially or in parallel, depend-

ing on a runtime flag.

Developed using C++, SPEEDES uses an object-oriented computational model. LPs are represented by simulation objects that are derived from the base class C_SIMOBJ. Each simulation object contains state data along with methods to access, modify, and analyze the data. Events are separate objects that are derived from the base class C_EVENT. Each event is associated with exactly one simulation object. In addition to exchanging data with that simulation object, an event can change its state or schedule other events.

To build a simulation program in SPEEDES, the user must define simulation objects, object managers for those objects, and events. An object manager class must be defined for each type of corresponding simulation object. For example, an airport object class (AIRPORT_OBJ) must have a corresponding airport manager class (AIRPORT_MGR). The object managers are responsible for creating and managing the set of simulation objects. Thus, the user (through the object managers) is *completely responsible* for the mapping of the simulation objects to the processors.

While SPEEDES gives the user freedom to choose an appropriate mapping, it is quite likely that the user does not know *a priori* how to choose a good allocation of objects to processors. The general mapping problem is NP-complete, so optimal mappings are extremely difficult to obtain. Furthermore, many users and potential users of PDES would prefer to let "the system" make such decisions, especially if the resulting performance is "good enough". In the next section, we describe the steps we took to automate the load-balancing process within SPEEDES so that the user can concentrate on the model rather than the implementation.

3 AUTOMATED LOAD BALANCING

Our work with SPEEDES began with developing a port for the Intel Paragon. Starting with the socket-based version for workstations, we rewrote the SPEEDES communication library to 1) take advantage of the Paragon's native communication and global reduction operations and 2) optimize the higher-level communication routines for the Paragon.

Once the port was complete, we modified SPEEDES to collect data on the workload characteristics of a simulation. Simulation objects determine the resolution of the partitioning since each event is connected to exactly one simulation object. To estimate the amount of computation required by a particular object, we modified the C_SIMOBJ base class to count the number of events that a simulation object processed and the number of those events that were committed. Notice that a large discrepancy in the numbers of the processed and committed events could indicate a load imbalance on the processor containing that object.

Next, we needed to determine the communication requirements between simulation objects. SPEEDES handles all communication (including the creation of events) through messages. We modified the C_SIMOBJ base class to keep counts of all messages sent to each destination object. The message counts include all messages that were transmitted, including antimessages.

The event and message counts are collected and saved in data files during a single run of the simulation. SPEEDES is instructed to collect the data (for one run) and use the data (in another run) for load balancing through the use of runtime flags. The data is analyzed by the mapping algorithm described below, that determines the load-balancing allocation of simulation objects to processors.

One may use the enhancements "as is" to gather information during a training run, and thereafter use a mapping based on the once collected run-time information. We see the ultimate application of the algorithm as part of a run-time remapping system, such as that reported in (Nicol and Mao, 1995). That algorithm first arranges the LPs in a linear chain, then partitions the chain in as many contiguous subchains as there are processors, mapping one subchain per processor. The partition chosen minimizes the amount of work assigned to the most heavily loaded processor, where the computation weight of a processor is the sum of measured computation weights of its LPs, and the computation weight of an LP is the number of committed events it executed. Given a linear ordering, this optimization problem can be solved very quickly, e.g. in $O(PM \log M)$ time, where P is the number of processors and M the number of LPs.

However, choice of an optimal linearization of LPs is an NP-complete problem; choice of a "good" linearization remains an open research problem. One idea is to linearize so as to keep heavily communicating LPs close to each other in the chain, thereby increasing the chance that they will be assigned to the same subchain. This intuition is is realized by a recursive heuristic which at the first step "pairs" LPs using a stable matching algorithm. Here the communication weight between two LPs is a measure of their attraction; a stable matching is one where if A and B are matched, and C and D are matched, it is not possible to break the matches and reassemble (e.g. A and C, B and D) and have higher attraction values for both matchings. Two LPs that are matched will

be adjacent to each other in the linear ordering. We then merge matched LPs into super-LPs, and compute the attraction between two super-LPs as the sum of the attractions between LPs in the two super-LPs. Matching these, the sets of LPs represented in two matched super-LPs will be adjacent to each other in the linear ordering. This process continues until there is a single super-LP. This process does not uniquely specify a linearization as it does not assign a left-right ordering to matched pairs. At present the left-right ordering is arbitrary.

The experiments reported in (Nicol and Mao 1995) use measured communication rates between LPs (messages per unit simulation time) as the base attraction function between LPs. This is suitable so long as there is only weak correlation between the communication between two LPs, and their computation weights. However, many simulations have "hot spot" simulation objects that perform most of the work. Typically, the hot spots have large amounts of communication with other hot spots. In our experience, using communication rates as the attraction function often clustered the hot spots together which resulted in poor load balancing. To "spread out" the hot spots, we adding large negative edge weights between hot spots to discourage such clustering. For the simulations we studied (described in the next section), this modification to the edge weights improved the balance of work and reduced execution times. Clearly more work is needed to find attraction functions that balance linearization's conflicting requirements of computational spread and low communication costs.

While the linear ordering determines the load-balanced mapping of simulation objects to processors, the SPEEDES user must choose to use that mapping when objects are created by the object managers. To assist the user, we added three functions to SPEEDES: `LB_is_avail()`, `is_local_object(objnum)`, and `is_local_object(objname)`. The `LB_is_avail()` function is used to determine if the load-balancing data is available for use during this run (i.e. it was collected during a previous run). Thus, the user can write an object manager that uses a default mapping if data is not available and the load-balanced mapping if it is. The `is_local_object(objnum)` function is used to determine if the simulation object with global ID number `objnum` should be created on this node while `is_local_object(objname)` provides the same information based on a user-defined object name. Notice that the automated mapping will be inappropriate unless the global ID numbers and object names are consistent from one run to the next.

4 RESULTS

We applied the automated load balancing in SPEEDES to two simulation models: a fully-connected queuing network and the DPAT model of the National Airspace System. In this section, we discuss the two models and present results from execution of the simulations on a 72-node Intel Paragon. Furthermore, we compare the results obtained from automated load balancing with those obtained from default partitioning and card dealing.

4.1 Qnet Simulation

Queuing networks are often used as PDES benchmarks because they can be difficult to simulate (Nicol 1988; Steinman 1991). Thus, we examined a fully-connected queuing network (Qnet) as the first test of our automated load-balancing system.

The Qnet simulation contained 1600 fully-connected servers, where each server was initially assigned 50 customers. As Steinman (1991) noted, a homogeneous network of queues can be quite uninteresting from a load-balancing viewpoint because "card dealing tends to work very well." Thus, we forced several queues to be "hot spots" to make the simulation more interesting. We examined three different Qnet scenarios.

1. 10 hot spots uniformly distributed among all 1600 servers, Prob(customer exiting queue needs service at hot spot)= .125
2. 10 hot spots uniformly distributed among the middle third of the servers, Prob(customer exiting queue needs service at hot spot)= .125
3. 20 hot spots uniformly distributed among all 1600 servers, Prob(customer exiting queue needs service at hot spot)= .250

Uniform distributions were used so that the customers did not show any preference for a particular server (hot spot or otherwise). Notice that the intensity of the hot spots (i.e. the amount of work each hot spot received) depended on the probability that an exiting customer (from any server) would go next to one of the hot spots.

For the simulations discussed in this paper, we used the optimistic Breathing Time Warp (BTW) synchronization protocol. Created by Steinman (1993), Breathing Time Warp combines the Time Warp and Breathing Time Buckets protocols. At the beginning of each global virtual time (GVT) cycle, messages are sent aggressively using Time Warp. Later in the cycle, all messages are sent risk-free using Breathing Time Buckets.

Two runtime parameters in SPEEDES determine the amount of risk in the BTW protocol: Nrisk and Nopt. For the first Nrisk events processed after the last GVT computation, messages are released immediately to the receiver (Time Warp). For the events from Nrisk to Nopt (where Nrisk < Nopt), event messages are cached locally and the event horizon is computed (Breathing Time Buckets).

The Breathing Time Warp parameters used for the Qnet simulation were Nrisk = 1500 and Nopt = 3000. These parameters were determined by conducting various runs of the Qnet simulation (using the default partitioning) on different numbers of processors. Overall, these parameters gave the shortest execution times.

For the Qnet simulation, the default mapping of simulation objects to processors used block partitioning. Specifically, n objects were allocated to p processors by placing the first n/p objects on the first processor, the next n/p objects on the second processor, and so forth. For comparison, we also examined the card-dealing approach in which the first object is placed on the first processor, the second object is placed on the second processor, etc., until all of the objects have been dealt to processors.

Figure 1 presents results for Qnet Simulation #1 in which 10 hot spots were uniformly distributed among the 1600 servers. For automated load balancing, we obtained the best results using committed event counts for computation weights and adding negative edge weights for hot spots to discourage clustering. For small numbers of processors, the execution times using default partitioning, card dealing, and automated load balancing were basically the same. For large numbers of processors, load balancing was the fastest while default partitioning was generally better than card dealing. Notice that when 64 processors were used, card dealing found a "lucky" mapping and thus matched the execution time obtained by load balancing.

Figure 2 presents results for Qnet Simulation #2 in which 10 hot spots were uniformly distributed among the middle third of the 1600 servers. Corresponding to the results from Simulation #1, load balancing gave the best execution times when the number of processors was large. Notice that the default (block) partitioning was the worst method. This poor performance is not surprising since the hot spots were clustered in the middle third of the objects and hence the workload was not evenly distributed. In this case, card dealing gave reasonable performance because it was able to break up the hot spots.

Figure 3 presents results for Qnet Simulation #3 in which 20 hot spots were uniformly distributed among the 1600 servers. In this case, the hot spots were "hotter" than the hot spots in the previous Qnet simulations, so default partitioning and card dealing had trouble with balancing the workload evenly among the processors. In this case, automated load balancing gave execution times that were much better than those obtained by the other two methods. When 12 or more processors were used, automated load balancing improved execution times by 22–77%.

It should be noted that when 64 processors were used, Qnet Simulation #3 using card dealing could not complete execution until the Nrisk parameter was reduced from 1500 to 150. In this case, the workload imbalance led to an avalanche of antimessages that slowed and eventually stalled the system after a few hundred seconds. The last data point for card dealing in Figure 3 was obtained using Nrisk = 150 while all of the other points were based on Nrisk = 1500.

For each of the Qnet simulations, automated load balancing made little or no improvement over block partitioning and card dealing when small numbers of processors were used. However, it gave good results when large numbers of processors were used, and the results were more consistent than those obtained by block partitioning or card dealing.

Given the uniformity of the communication patterns (excluding the hot spots), the interobject communication was probably not much of a factor for this simulation. Instead, the automated load balancing took advantage of its ability to examine the computation weights of each object while the other methods examined only object counts.

4.2 DPAT: A Model of the National Airspace System

For the last several years, the MITRE Corporation has been studying the National Airspace System (NAS), which encompasses all commercial and general aviation air traffic in the United States (Wieland, Blair, and Zukas 1995). On a typical day, the NAS consists of 45,000 to 50,000 flights from approximately 16,000 airfields. The commercial air traffic is handled by roughly 1000 airports while 80% of the general aviation traffic is handled by the top 500 airports. In addition to the airfields, the NAS contains 701 three-dimensional regions called sectors that cover the airspace between airports.

MITRE recently developed a PDES model of the NAS called DPAT (Detailed Policy Assessment Tool) that is used to examine the average delay encountered by aircraft under various weather and traffic conditions. As discussed by Wieland, Blair, and Zukas (1995), the physical NAS system is a good candidate

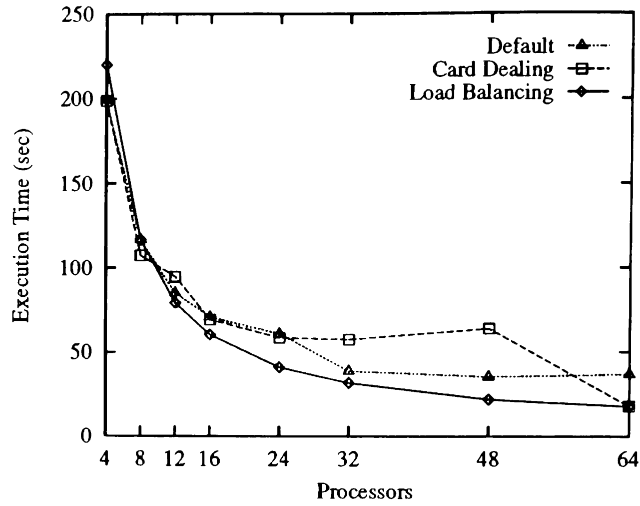

Figure 1: Results for Qnet Simulation #1 (10 hot spots uniformly distributed)

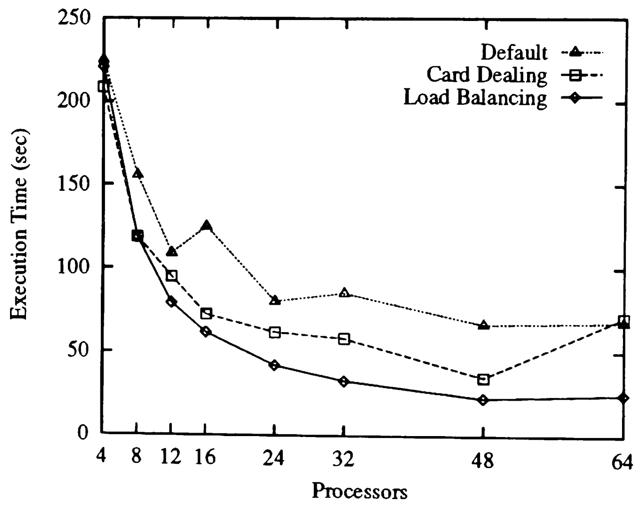

Figure 2: Results for Qnet Simulation #2 (10 hot spots in middle third)

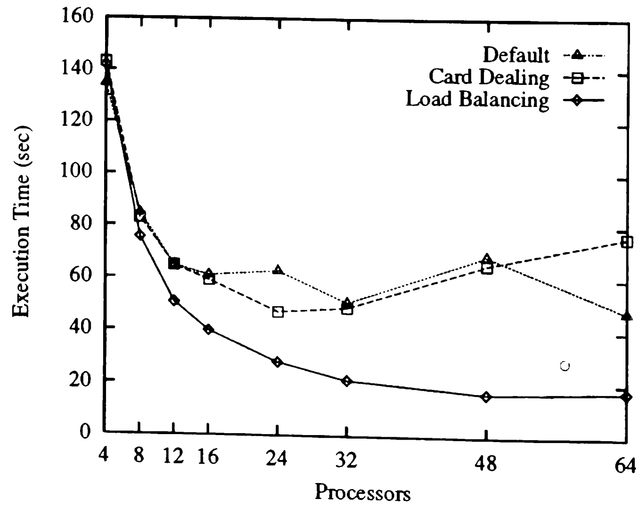

Figure 3: Results for Qnet Simulation #3 (20 hot spots uniformly distributed)

for PDES because the aircraft, air traffic controllers, and airports operate naturally in parallel.

The DPAT model contains SPEEDES simulation objects for 520 airports and 701 sectors. Events in the system include takeoffs, landings, and transfers of aircraft between sectors. Scheduling data from the Official Airlines Guide (OAG) is used to schedule commercial flights while general aviation flights are scheduled stochastically. Details of this model can be found in (Wieland, Blair, and Zukas 1995).

The DPAT simulation begins by reading in large files of flight and airplane data to initialize system parameters and schedule initial events. When we first executed DPAT on the Intel Paragon, we encountered severe performance problems due to memory paging. In particular, the aggregate size of the executable and data exceeded the roughly 23 MBytes per node of user-available memory. After discussing the problem with MITRE, we modified the program to use a subset of the aircraft data. This modification eliminated the memory problems (when multiple processors were used) without reducing the amount of computational work required.

The DPAT simulation organizes the airports and airspace sectors into geographic groupings called centers. For example, the La Guardia, Kennedy, and Newark airports in New York and New Jersey belong to a center that contains the airspace sectors around those airports. The Los Angeles and San Francisco airports belong to different centers because of the distance (and number of other airports) between them. These geographic centers form the basis for DPAT's default partitioning of simulation objects to processors.

Given that a flight must travel through contiguous sectors between airports, it is logical to assume that a geographical partitioning of the airports and sectors will reduce communication costs. The problem, however, is that the geographic distribution may not result in an even distribution of work. With DPAT, the airports and sectors are divided among 22 centers, where the "laziest" center is associated with 421 events and the busiest center has 20963. Even if more than 22 processors are used, only 22 will receive work. This center-based approach serves as the default mapping for DPAT.

As an alternative, the card-dealing approach assigns an equal number of objects to each processor without regard for interobject communication. Furthermore, card dealing has no knowledge of relative weights of the objects, so the work may not be evenly distributed after all.

We executed DPAT on the Intel Paragon using three different mappings: default (center-based), card dealing, and load balancing. For consistency, all of the runs were taken with Nrisk = 250 and Nopt = 500, which were determined from the best timings of the default partitioning.

Figure 4 presents results for the DPAT simulation under the default (center-based) allocation, card dealing, and load balancing. It is easy to see that the default mapping gave the worst results. However, we were surprised to see that card dealing and automated load balancing gave very similar results for up to 24 processors. It appears that the wide variety of computational weights for the objects made it easy for card dealing to find fairly even distributions of work. With very large numbers of processors (32 to 64), automated load balancing was better than card dealing by 20–25%. Thus, automated load balancing in SPEEDES can give significant improvement in execution times when large numbers of processors are used.

5 CONCLUSIONS

Given the simplicity of our approach, it seems to do a good job! As expected, there is clearly benefit to using run-time information in a fast, simple algorithm to guide mapping. In general, card dealing provides a good initial mapping for the collection of run-time data. For further work, we will complete the cycle, giving SPEEDES the ability to migrate objects to implement at run-time the mappings based on run-time information.

ACKNOWLEDGMENTS

This work was supported by the National Aeronautics and Space Administration under NASA Contract NAS1-19480 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, VA 23681. Professor Nicol's work is also supported in part by NSF Grant CCR-9201195.

We would like to thank Jeff Steinman and Fred Wieland for their assistance with SPEEDES and DPAT.

REFERENCES

Nicol, D. M. 1988. Parallel Discrete-Event Simulation of FCFS Stochastic Queuing Networks. *Proceedings ACM/SIGPLAN PPEALS 1988: Experiences with Applications, Languages and Systems*, pp. 124–137.