A distributed discrete event simulation,

or How to steal more CPU cycles than you could ever imagine

Asbjoern M. Bonvik

June 2, 1995

Discrete event simulation is, almost by definition, a very CPU-intensive task. Some stochastic process is represented by a program incorporating a pseudo-random number generator, and statistics are collected by executing the program. It may be necessary to run the simulation for a very long time to get good estimates of the performance measures of the actual system under study. For example, if one is studying a manufacturing system where machine failures are important to the overall behavior of the system, the simulation needs to cover a period with many failures of different lengths.

It is very hard to parallelize a discrete event simulation, because the commonly found algorithm maintains a centralized event list and a current simulation time, executing one event at a time. The current event may manipulate the list of future events, for instance by scheduling the end of some activity or the next instance of some event. When the current event is done, the event with the lowest activation time is taken out of the event queue, the simulation time is set to this value, and the event is executed. This essentially presumes a single thread of execution on a single processor.

On the other hand, because of statistical considerations, it is common to split a large simulation job into a batch of independent *replications*, where each replication is started from a different random number seed, everything else being the same. This is done to eliminate the dependency on a single sequence of pseudo-random numbers and to establish confidence intervals on the performance measures. This also gives a natural granularity for a coarse-grained parallel or distributed computation: Distribute one replication to each processor, and designate one processor to collect the statistics emanating from the computation.

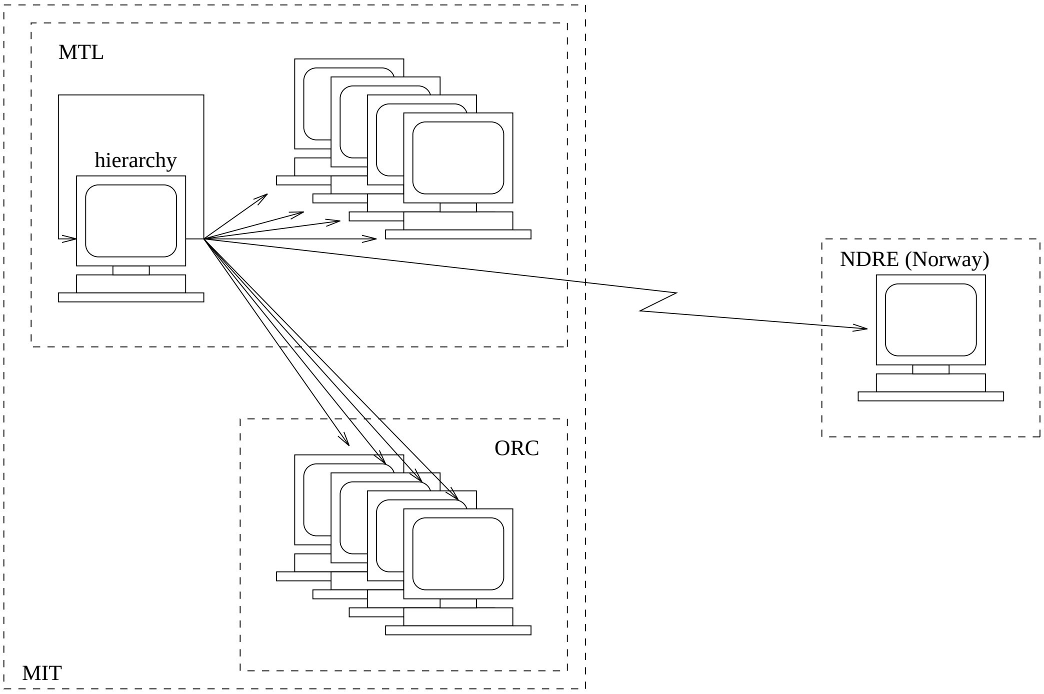

Figure 1: Distributed simulation architecture.

This is exactly what we did for a recent study of a small production system under various conditions. Our work had two phases; one screening phase where a large number of parameter settings had to be evaluated, and one refinement phase where multiple replications were done of the best parameter settings to draw final conclusions. To do this efficiently, we constructed a distributed simulator architecture that ran on about one dozen Sun Sparcstations, mainly SparcStation 10s. The computers used were the MTL machines, the machines at the Operations Research Center, and a single SparcStation 10 at the Norwegian Defence Research Establishment. Figure 1 shows the general topology. After the actual study was completed, a Pentium 100MHz PC running Linux was added to the architecture. This demonstrates that the architecture is not Sun-specific. Incidentally, the PC turned out to work some 20% faster than even a SparcStation 20, possibly because of faster I/O leading to less overhead when starting a client.

The software had three components: the discrete event simulation program itself, a client program that drove a single replication on some host, and the server program that organized the computation and collected statistics.

The discrete event simulation is a relatively small and efficient C program based on the CAFE scheduling and simulation testbed. This version was rewritten for efficiency in time and memory, so a lot of the more impressive features of the testbed were left out. It did, on the other hand, include more statistics collection and reporting than the testbed. The resulting program occupied about 140K of memory (which is smaller than a single `csh` process), and did very little input/output. The input files were about 250 bytes and the output 2500 bytes per replication. Run times depended strongly on the input, but 100 weeks of simulated time tended to take about 5 minutes on a SparcStation 10.

The client and server applications were written in Perl. The server starts up one replication on each host (including the one it is executing on, which we will call the *master host*) by running a `rsh` ***`hostname`*** `client` with standard input and standard output redirected to local files on the master host. The client and simulation are executed with priority 10, which minimizes interference with interactive users and other jobs. The server stores the process id number of the `rsh` process in an associative array indexed by hostname. In addition to simulation control parameters, the command line arguments passed to the client include the name of the master host and a port number.

To use `rsh` in this way, it is necessary to have an account on each host computer, and in the home directory of that account have a file `.rhosts` that allows the user to log in from the master host. See `man rsh` for details.

After starting one job on each host, the server listens for completion notices on a socket connected to this port. The completion notice is written to this socket by the client when the actual simulation is done, and consists of the client host name and a status word. After that, the client exits, and the `rsh` process also terminates. The server will then know that the client on the named host has completed (successfully or otherwise) and will parse the resulting output to update overall statistics. It then launches another client to the host as needed.

The other main functionality in the client/server interaction is signal handling. If the server is killed (for instance if the user hits Control-C), this signal will be passed on to the client, and appropriate house cleaning is done before termination.

The server comes in two flavors, `pgstry` and `psgrun`. The first one is for running a single replication of each of a number of input files. A typical invocation is `pgstry *.inp`. The server launches a job corresponding to an input file on each host, and channels the results to a file where the extension

`.inp` is replaced by `.gso`. When one job finishes, another job is directed to that host. Currently, not much effort has gone into error handling and recovery for this version, so if some host crashes, no attempt is made to restart that job on some other host. Nor has there been any effort to detect host workload and speed to optimize the job allocation over hosts.

The other version runs multiple replications of a single input file, invoked as `pgsrun -N 150 myfile.inp`. This program will run until the number of completion notices asked for (150 in this case) has been received. It then kills all remaining processes and terminates. This program is more robust in the face of host failures and other disturbances, since as long as completion notices arrive at the server from some host, the entire job will finish. If some client is terminated abnormally, for instance because of a system crash, it will not send a completion notice, and no more jobs will be sent to that host.

Both server versions have some degree of automatic load balancing between hosts, since job allocations are made when earlier jobs are completed. This will lead to a slow host receiving fewer jobs than a fast one, and if the apparent speed changes due to workload fluctuations during a run, the workload allocation will shift to reflect this.

There is a tradeoff between using few and long simulations versus many short ones. The load balancing tends to work better with many jobs, since there is less chance of a "slow" host holding up the rest of the computation (an issue in `pgstry` only). On the other hand, a new client and simulation program needs to be launched for each new job, so the administrative overhead increases with the number of jobs. A tradeoff was made by trial and error.

By using these programs, we successfully simulated about 6000 different cases. Most cases were run for a single replication of about 100 weeks of simulated time, but about 50 cases were selected for further scrutiny. This required 50 more replications of each case, for a total of about 8500 replications. We estimated this to take about 3 months of CPU time on a single SparcStation 2 such as hierarchy. Using our virtual supercomputer, the entire job was done in four days of real time. In addition, we did extensive model validation runs and a number of false starts before getting it right.

More speed is available by using more hosts. The approach is not limited to Sun computers,as our Pentium PC demonstrated. Anything that supports UNIX stream sockets can be used. This allows other fast computers to be added by the simple expedient of getting an account, recompiling the simulation for that machine type, and adding the host name to the server list of target hosts.

In conclusion, our approach gives supercomputer speed at very little cost. Suitable jobs for a similar scheme are very CPU-intensive with small memory and I/O requirements, and have a structure such that fairly large pieces of the job can be computed separately and the results assembled with little effort.