How Are We Doing? An Efficiency Measure for Shared, Heterogeneous Systems

R.D. Chamberlain
D. Chace
A. Patil

R.D. Chamberlain, D. Chace, and A. Patil, “How Are We Doing? An Efficiency Measure for Shared, Heterogeneous Systems,” in *Proc. of the ISCA 11th Int'l Conf. on Parallel and Distributed Computing Systems*, September 1998, pp. 15-21.

Washington University
and
Southern Illinois University Edwardsville

How Are We Doing? An Efficiency Measure for Shared, Heterogeneous Systems

R. Chamberlain*, D. Chace†, and A. Patel*

*Dept. of Electrical Engineering
Washington University
St. Louis, MO

†Dept. of Electrical and Computer Engineering
Southern Illinois University Edwardsville
Edwardsville, IL

Abstract

The utilization of networked, shared, heterogeneous workstations as an inexpensive parallel computation platform is an appealing concept that has received quite a bit of attention in the literature. However, most efficiency measures for parallel computation are oriented towards the use of tightly-coupled, dedicated, homogeneous processors. We propose a new efficiency measure for parallel computations executing on networks of workstations and illustrate this efficiency measure on an example computation.

1 Introduction

An important consideration in any engineering design effort is how to judge the system that is being designed. This is important for a number of reasons both during the design process (e.g., to enable designers to choose between competing design options) and once the system has been built (e.g., to enable comparisons between the system and other systems with similar goals). In parallel processing systems, the gold standard judgment mechanism (or figure of merit) has been speedup. Essentially, how much faster does the parallel system solve a problem than a fairly-chosen uniprocessor system. This is typically quantified as follows: the speedup on P processors, $S(P)$, is the ratio of the execution time on 1 processor, $T(1)$, to the execution time on P processors, $T(P)$.

$$S(P) = \frac{T(1)}{T(P)} \tag{1}$$

(Note, the symbols used are summarized in Table 1.) Intuitively, this definition makes a lot of sense. The major motivation for parallelism is to execute codes faster, and speedup is a direct measure of how successful we have been at accomplishing that goal.

In a perfectly parallelized system, one would expect the parallel execution to be P times as fast as the uniprocessor execution, with a resulting speedup of $S(P) = P$. This upper bound on speedup is often referred to as linear speedup, and anything smaller is called sub-linear speedup. (The names have persisted in spite of the obvious conflict with the usual definitions of linear and sub-linear.) In this context, parallel efficiency is defined as

$$\eta(P) = \frac{S(P)}{P} \tag{2}$$

and our usual expectations for an efficiency measure apply (i.e., it ranges from 0 to 1, low values represent low utilization of computational resources, high values represent high utilization of resources, etc.). This definition is illustrated in Figure 1, which plots a linear speedup curve (dashed line) and a sub-linear speedup curve (solid line). The parallel efficiency of the sub-linear speedup curve is the ratio a/b.

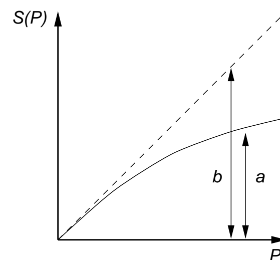

Figure 1: Example speedup curve

There are a number of issues that must be considered when applying a speedup metric to a parallel system. First, one must fairly choose the algorithm and implementation used to determine the uniprocessor execution time. This should be the fastest known uniprocessor algorithm, which is often not simply the 1 processor implementation of the parallel algorithm. Note that longer uniprocessor execution times increase

the apparent speedup, without actually making the parallel system any faster.

Second, when parallelism provides performance improvements on some problems, it is common to trade off faster runtimes for the ability to solve larger problems. In other words, the user is more interested in scaling up the problem size than decreasing the execution time. This is often the case when the problem is memory limited (i.e., does not fit in the available memory on one processor). By increasing the number of processors, the available memory increases as well. A common figure of merit used to judge these conditions is scaled speedup. Here, the problem size is scaled up with the number of processors. Since the workload is scaled up by a factor of P, it is customary to scale the speedup measure as well [1]. Scaled speedup is defined as

$$S_{SC}(P) = \frac{PT(1)}{T(P)} \quad (3)$$

and scaled efficiency is scaled speedup divided by P:

$$\eta_{SC}(P) = \frac{S_{SC}(P)}{P} = \frac{T(1)}{T(P)} \quad (4)$$

It is also possible to generalize the connection between scaled problem size and efficiency. For a parallel system (defined as a parallel algorithm and the parallel architecture on which it is executed), the isoefficiency is the rate at which the problem size must be scaled (as the number of processors is scaled) to maintain a constant efficiency [2].

There are several assumptions, however, that are implicit in the above definitions. Speedup (1) and parallel efficiency (2) became popular figures of merit when parallel processing was constrained to specialized hardware that typically executed one parallel program at a time. All of the processing nodes were alike (i.e., same CPU, clock rate, memory subsystem, etc.), they were connected via a dedicated, regular interconnection network (e.g., shared bus, mesh, hypercube, etc.), and the operating system was either single-tasking or at least single-user (a program had exclusive access to the computational resources for the duration of the execution).

More recently, parallel processing techniques are being applied to networks of existing workstations. The majority of the time, these workstations are idle and therefore under-utilized. The processing environment assumed here is a network of workstations that communicate via a local-area network (see Figure 2). The workstations are not dedicated resources; several users may be utilizing them while the computation of interest is executing. In addition, the power (i.e., computational speed) of the individual workstations may vary, although we will assume their basic architecture is the same (single CPU, significant local memory, possibly local disk). In order to execute parallel codes on the workstations, there are a number of systems available that provide message passing and process control primitives [3]. Our experimental results use the PVM system [4].

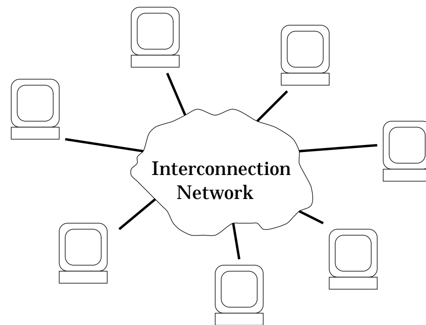

Figure 2: Network of workstations

The fact that not all of the processors are alike poses significant difficulties with the above figures of merit. Both the speedup measure (1) and parallel efficiency (2) assume that the available computational power associated with a processor is both uniform and constant in time. The heterogeneous nature of workstations on a local-area network violates the uniformity assumption, and the shared nature of the workstations implies computational power that varies with time (as other users compete for the workstations cycles).

One approach, which does address the heterogeneity issue, is to sort the processors by their computational power, and measure speedup relative to the most powerful processor. If we denote the rate at which computation proceeds on processor p as $R(p)$, then $R(i) \geq R(j)$ for $i \leq j$. Here, the maximum possible speedup, $S_M(P)$, is calculated for P processors as

$$S_M(P) = \left(\sum_{p=1}^{P} R(p) \right) / R(1). \quad (5)$$

Efficiency can be defined as the ratio of the achieved speedup to the maximum possible speedup.

$$\eta_H(P) = \frac{S(P)}{S_M(P)} \quad (6)$$

Note that for the homogeneous case, $R(i) = R(j)$ for $1 \leq i,j \leq P$, the maximum possible speedup is P, and

(6) reduces to (2). This approach does not, however, deal with the case of time varying resources.

This paper defines a new figure of merit for comparing parallel program executions, called shared parallel efficiency. This figure of merit describes how effective a parallel code is at utilizing the resources that are available, even when they are shared, heterogeneous processors. We illustrate the use of the figure of merit, and describe some of the issues that surface when performing empirical experiments to measure the shared parallel efficiency of a running program.

Table 1: Symbol definitions

Symbol	Meaning
$C(p;t_1,t_2)$	Compute cycles available on processor p over time interval (t_1,t_2)
$N(p,t)$	Number of background tasks on processor p at time t
$R(p)$	Compute rate available on processor p when unloaded (dedicated to execution of parallel code)
$R_a(p,\tau)$	Actual compute rate achieved on processor p over time interval τ
$R_e(p,\tau)$	Effective compute rate available on processor p over time interval τ
$R_A(P,\tau)$	Actual total compute rate achieved on P processors over time interval τ
$R_E(P,\tau)$	Effective total compute rate available on P processors over time interval τ
$S(P)$	Speedup as traditionally defined on P processors
$S_M(P)$	Maximum speedup on P heterogeneous processors
$S_{SC}(P)$	Scaled speedup on P homogeneous processors
$T(P)$	Time required to execute parallel code on P processors
$\eta(P)$	Traditional parallel efficiency on P processors
$\eta_H(P)$	Parallel efficiency on P heterogeneous processors
$\eta_{SC}(P)$	Scaled efficiency on P homogeneous processors
$\eta_{SH}(P,\tau)$	Shared parallel efficiency on P processors over time interval τ.

2 Shared Parallel Efficiency

The sharing of any computational resource involves a number of policy questions. Primarily, the issues revolve around who has priority in the use of the computational cycles available on a machine. At one end of the spectrum, we might assume that the parallel computation has priority, and other users are simply inconvenienced whenever the parallel code is executed. At the other end of the spectrum, the parallel code might have the lowest priority, in an attempt to minimize its impact on the other users of the machine (e.g., this is the policy implemented in the Condor system [5]). We are interested in a middle ground, where the parallel program and the other load on the workstation are at roughly equal priorities, effectively sharing the computational resources.

In this case, an individual processor p with an average background load of N tasks (prior to the introduction of the parallel application) will have an effective computation rate of $R(p)/(N+1)$ available to the parallel code, where $R(p)$ is as defined above (the computation rate available on processor p when there is no background load). In general, the background load on processor p is a function of time, $N(p,t)$, and the total number of cycles available to the parallel program over a time interval t_1 to t_2 is the integral of the instantaneous effective computation rate

$$C(p;t_1,t_2) = \int_{t_1}^{t_2} \frac{R(p)}{N(p,t)+1} dt. \tag{7}$$

The overall effective computation rate that is available to the parallel application is

$$R_e(p,\tau) = \frac{C(p;t_1,t_2)}{t_2-t_1}, \tag{8}$$

where τ represents the time interval (t_1,t_2).

R_e represents the best any parallel execution could possibly do, limited only by available CPU cycles. As such, it is a good candidate for the denominator of the shared parallel efficiency metric. For the numerator, we simply use the computation rate actually achieved on processor p over time interval τ, denoted $R_a(p,t)$. Hence, the shared parallel efficiency for an individual processor is R_a/R_e.

What remains is the need to combine individual processor efficiencies into a total efficiency metric. To accomplish this we combine the effective available rates on each processor into a total effective compute rate available on the set of processors P.

$$R_E(P,\tau) = \sum_{p=1}^{P} R_e(p,\tau) \tag{9}$$

We also combine the actual compute rates achieved into a total compute rate achieved on the set of P

processors.

$$R_A(P,\tau) = \sum_{p=1}^{P} R_a(p,\tau) \tag{10}$$

The shared parallel efficiency is then

$$\eta_{SH}(P,\tau) = \frac{R_A(P,\tau)}{R_E(P,\tau)}. \tag{11}$$

The denominator for shared efficiency on a single processor is illustrated in Figure 3(a), which shows an example background load profile over a range of time. Here, the computational rate of the processor is plotted as the flat line $R(p)$, and the background load is represented by the shaded region of the graph. The unshaded region between the background load and the maximum computation rate represents the computational cycles available to the parallel program, $C(p; t_1, t_2)$. The average rate at which these cycles are made available corresponds to $R_e(p,\tau)$.

Figure 3(b) expands this example to include a parallel program consuming cycles at an instantaneous rate denoted by the dark shaded region on the graph. The average value of this rate is $R_a(p,\tau)$. As a result, the shared efficiency associated with this processor is the ratio of the dark shaded area of Figure 3(b) to the unshaded area of Figure 3(a).

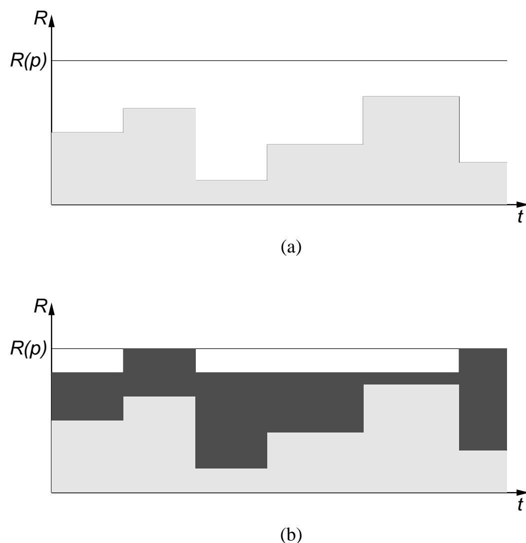

Figure 3: Shared parallel efficiency illustration

One of the implications of the above definitions is the implicit equivalence between a lightly-loaded, slower processor and a heavily-loaded, faster processor. The R_e for an unloaded processor is equal to the R_e for a processor that is twice as fast but has a background load that consumes 50% of the cycles. The effective computation rate available to a parallel program is the same on these two processors, and a good workload partitioning algorithm would assign an equal amount of work to each.

3 Examples

In this section, we describe a number of experiments designed to illustrate the definition of shared parallel efficiency. We first describe the parallel program used in the experiments. We then illustrate the shared parallel efficiency metric when used on a homogeneous processor set (both with and without background load) and a heterogeneous processor set (again, both with and without background load).

3.1 Parallel Program

The parallel program used to illustrate shared parallel efficiency is a Monte Carlo simulation used to estimate the value of π. If a circle with radius one is centered in a square whose sides are length two (see Figure 4), the area of the circle is π and the area of the square is 4. In the Monte Carlo simulation, darts are thrown at the square, uniformly distributed across the area of the square, and the darts that land inside the circle are counted. The ratio of darts in the circle to total darts is then $\pi/4$.

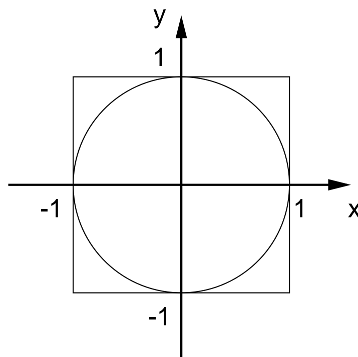

Figure 4: Calculation of π

Due to the symmetry in the geometry, it is reasonable to constrain the simulation to the first quadrant ($x \geq 0, y \geq 0$). The pseudo code for the simulation is given in Figure 5.

The parallel simulation code is organized in a master-slave arrangement, with the master allocating workload to the slaves by assigning a varying number of darts for simulation to each slave. Currently, the workload distribution is guided manually. Once the

```
MCsim (number_of_darts)
      hits = 0
      for i = 1 to number_of_darts
            x = uniform_dist(0,1)
            y = uniform_dist(0,1)
            if y < sqrt(1 - x^2) then
                  increment hits
            endif
      endfor
      return(hits)
```

Figure 5: Monte Carlo simulation

slaves complete their assigned work, the master checks for convergence and, if necessary, repeats the main execution loop, assigning additional darts to each slave.

This parallel program clearly has a high degree of scalability. There is very limited interprocessor communication, and the computation is heavily CPU-bound. These properties make it ideal for illustrating the shared parallel efficiency performance metric, since (with an appropriate workload distribution) the efficiency can approach 1.

In order to perform controlled experiments, the background load is synthetically generated. It consists of calculating the FFT of an image (chosen large enough to force the example program out of cache). To further limit uncontrolled effects, it is executed 50% of the time at a sufficiently high priority to ensure that it consumes very close to 50% of the CPU cycles. This is essentially the same synthetic background load generator as used in [6].

3.2 Experiments

The first set of experiments used 2 identical machines connected via an Ethernet network. When unloaded, each of these machines executes the Monte Carlo simulation at a rate of $R(p) = 30,000$ darts/sec, $1 \leq p \leq 2$ (measured using uniprocessor runs). To confirm the small parallel overhead, the first run used an equal workload distribution on the two processors (without any background load), and achieved an overall execution rate of $R_A = 59,900$ darts/sec, corresponding to a shared parallel efficiency of $\eta_{SH} = 0.998$.

In the second experiment, processor $p = 2$ was given a synthetic background load (described above) that consumed 50% of the processor's cycles. This corresponds to an average background load of $N = 1$ and an effective available computation rate of $R_e(2,\tau) = 15,000$ darts/sec. The total available computation rate is now $R_E = 45,000$ darts/sec (processor $p = 1$ has no background load, so $R_e(1,\tau) = 30,000$ darts/sec). As in the first experiment, an equal workload distribution was used across the two processors, and an overall execution rate of $R_A = 29,950$ darts/sec was achieved. This corresponds to a shared parallel efficiency of $\eta_{SH} = 0.666$. Essentially, one third of the available computational resources (one half of the cycles on processor 1) went idle.

For the third experiment, the background load was kept the same as experiment 2, and the workload distribution was changed to assign 2/3 of the work to processor 1 and 1/3 of the work to processor 2. Here, the total available computation rate is unchanged from experiment 2, but the overall execution rate achieved was $R_A = 44,400$ darts/sec. The shared parallel efficiency is now $\eta_{SH} = 0.987$, reflecting the better workload distribution. The empirical results for the first set of experiments are summarized in Table 2.

The second set of experiments used 2 heterogeneous machines connected via an Ethernet network. When unloaded, processor 1 executes the Monte Carlo simulation at a rate of $R(1) = 30,000$ darts/sec and processor 2 runs at $R(2) = 85,000$ darts/sec (again measured using uniprocessor runs). Experiment 4 repeats the even workload assignment and no background load conditions of experiment 1. In this case, $R_A = 59,300$ darts/sec, $R_E = 115,000$ darts/sec, and $\eta_{SH} = 0.516$. Clearly, processor 1 is the limiting factor in the overall performance.

Experiment 5 adds a 50% background load to processor 2, lowering its available compute rate to $R_e(2,\tau) = 42,500$ darts/sec. With an even workload distribution, however, processor 1 is still the limiting factor in execution speed, and the resulting compute rate is nearly identical at $R_A = 59,600$ darts/sec.

Experiment 6 goes back to the no background load case, and redistributes the workload, 1/3 to processor 1 and 2/3 to processor 2. Here, since more of the work is assigned to the faster processor, the compute rate goes up to $R_A = 88,900$ darts/sec, and the shared parallel efficiency is $\eta_{SH} = 0.773$.

Experiment 7 keeps the workload distribution of experiment 6, and adds the 50% background load to processor 2. This time, processor 2 is the performance limiting factor, the overall compute rate is $R_A = 63,800$ darts/sec, and the shared parallel efficiency is $\eta_{SH} = 0.880$. The results for the second set of experiments are summarized in Table 3.

The goal of the shared parallel efficiency metric is to accurately reflect how well the parallel program is using the resources available. The above examples illustrate several cases of both efficient resource utilization

Table 2: Homogeneous processors

Experiment	Processor 1		Processor 2		R_E	R_A	η_{SH}
	$R_e(1,\tau)$	workload	$R_e(2,\tau)$	workload			
1	30,000	50%	30,000	50%	60,000	59,900	0.998
2	30,000	50%	15,000	50%	45,000	29,950	0.666
3	30,000	67%	15,000	33%	45,000	44,400	0.987

Table 3: Heterogeneous processors

Experiment	Processor 1		Processor 2		R_E	R_A	η_{SH}
	$R_e(1,\tau)$	workload	$R_e(2,\tau)$	workload			
4	30,000	50%	85,000	50%	115,000	59,300	0.516
5	30,000	50%	42,500	50%	72,500	59,600	0.822
6	30,000	67%	85,000	33%	115,000	88,900	0.773
7	30,000	67%	42,500	33%	72,500	63,800	0.880

and inefficient resource utilization, and the figure of merit accurately reflects the quality of the utilization.

4 Efficiency Measurement

The above experiments, used to illustrate the shared parallel efficiency performance metric, are limited in the fact that they were executed under controlled conditions. Rather than reacting to the background load that happened to be present on the machines, a synthetic background load (with known properties) was created and executed. In this case, we know the properties of the background load, and use that knowledge in the calculation of the shared parallel efficiency. For the shared parallel efficiency metric to be useful in practical situations, one must be able to empirically measure the value of the metric on an executing program, with an uncontrolled background load. In this section, we describe our initial attempts at achieving this goal.

In Unix systems, the load average is defined as the average number of jobs in the run queue over some time interval, or N. We hypothesize, therefore, that the load average should correlate well with the time required to execute a serial, CPU-bound program. If this is the case, then the load average can be used to quantify the background load present during the execution of a parallel program, and can lead to an empirical measurement of shared parallel efficiency in an uncontrolled background load environment. To test this hypothesis, the following experiment was performed (in parallel) on a number of processors.

1. Measure the 1 minute load average of the target processor (using uptime).

2. Measure the execution time of a simple busy-loop program.

3. Wait for a period of time long enough that the above 2 steps don't impact a subsequent load average measurement ($>$ 1 minute).

4. Repeat from step 1.

Each loop above generates a pair of numbers (load average, execution time) that should be highly correlated if the hypothesis is true. The execution of the sample program occurs following the load average measurement so that the load average is not impacted by the program. The delay ensures that each measurement is independent (i.e., not impacted by the previous loop).

A scatter plot of the results (plotting execution time vs. load average) for one of the processors is shown in Figure 6. As is readily apparent, although there is a correlation between load average and execution time, it is not a tight one, and the variation is quite high. In addition, this plot represents the best correlation detected. For most of the processors, the results were worse yet. The conclusion we draw from this is that load average (as reported by the system) is inadequate for the task of measuring the background load present during a parallel program execution. We are currently investigating several other methods for effective empirical measurement of background load.

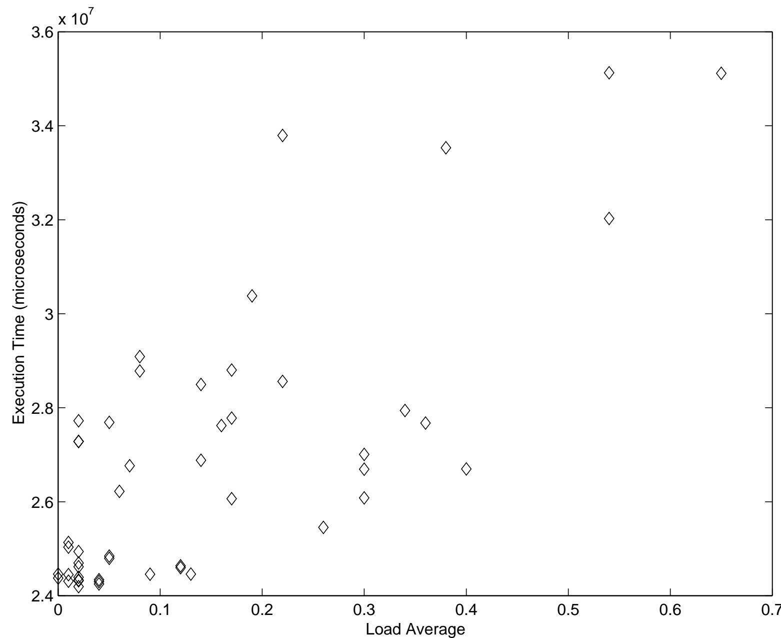

Figure 6: Background load measurement experiment

5 Summary and Conclusions

This paper proposes a new figure of merit for judging the effectiveness of parallel program executions in a shared, heterogeneous resource environment. The shared parallel efficiency metric addresses the limitations inherent in traditional efficiency measures when presented with a time-varying computational platform. When used in a dedicated, homogeneous resource environment, the shared parallel efficiency is equal to the standard definitions. Hence, it is a generalization of the traditional measure.

The shared parallel efficiency of an example parallel program was illustrated on a homogeneous processor set and a heterogeneous processor set, both with and without background load. In addition, the system reported load average was investigated as a method for empirically measuring background load.

Once an effective method for experimental determination of background load is in place, the shared parallel efficiency metric will provide a uniform, consistent mechanism for comparing parallel algorithms, workload distribution algorithms, and a host of other issues that arise when executing parallel programs on shared, heterogeneous processors.

Acknowledgements

The research described here was supported in part by the National Institutes of Health under grant GM28719, the National Science Foundation under grant CTS-9612499, and the Undergraduate Research Academy at Southern Illinois University Edwardsville.

References

[1] J.L. Gustafson, “Reevaluating Amdahl’s Law,” *Communications of the ACM*, 31(5):532–533 (1988).

[2] A. Grama, A. Gupta, and V. Kumar, “Isoefficiency Function,” *IEEE Parallel and Distributed Technology*, 1(3):12–21 (1993).

[3] L.H. Turcotte, “A Survey of Software Environments for Exploiting Networked Computing Resources,” Tech. Rep. MSU-EIRS-ERC-93-2, NSF Engineering Research Center for Computational Field Simulation, Mississippi State University, Starkville, MS (1993).

[4] V.S. Sunderam, “PVM: A Framework for Parallel Distributed Computing,” *Concurrency: Practice and Experience*, 2(4):315–339 (1990).

[5] M. Litzkow and M. Livny, “Experience With the Condor Distributed Batch System,” in *Proc. IEEE Workshop on Experimental Distributed Systems* (1990).

[6] B.L. Noble and R.D. Chamberlain, “Performance of Speculative Computation in Synchronous Parallel Discrete-Event Simulation on Multiuser Execution Platforms,” in *Proc. 8th IASTED Int’l Conf. on Parallel and Distributed Computing and Systems*, pp. 489–494 (1996).