Cost/Performance of a Parallel Computer Simulator*

Babak Falsafi and David A. Wood

Computer Sciences Department
University of Wisconsin–Madison
1210 West Dayton Street
Madison, WI 53706 USA
wwt@cs.wisc.edu

Abstract

This paper examines the cost/performance of simulating a hypothetical *target* parallel computer using a commercial *host* parallel computer. We address the question of whether parallel simulation is simply faster than sequential simulation, or if it is also more cost-effective. To answer this, we develop a performance model of the Wisconsin Wind Tunnel (WWT), a system that simulates cache-coherent shared-memory machines on a message-passing Thinking Machines CM-5. The performance model uses Kruskal and Weiss's fork-join model to account for the effect of event processing time variability on WWT's conservative fixed-window simulation algorithm. A generalization of Thiebaut and Stone's footprint model accurately predicts the effect of cache interference on the CM-5. The model is calibrated using parameters extracted from a fully-parallel simulation ($p = N$), and validated by measuring the speedup as the number of processors (p) ranges from one to the number of target nodes (N). Together with simple cost models, the performance model indicates that for target system sizes of 32 nodes and larger, parallel simulation is more cost-effective than sequential simulation. The key intuition behind this result is that large simulations require large memories, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory.

*This work is supported in part by NSF PYI Award CCR-9157366, NSF Grant MIP-9225097, and donations from Thinking Machines Corp., Xerox Corp., and Digital Equipment Corp. Our Thinking Machines CM-5 was purchased through NSF Grant No. CDA-9024618 with matching funding from the Univ. of Wisconsin Graduate School.

1 Introduction

The architecture of a parallel computer specifies an interface between software and hardware. Computer architects prefer to study the complex interactions across this interface by running and measuring real applications. Simulation allows evaluation of these interactions without building hardware prototypes, speeding the design process.

Simulation has long been used to evaluate proposed computer hardware for correctness and performance. However, most simulations have focussed on low-level implementation details: circuit level, switch level (ideal transistor), or logic (gate) level. These detailed simulations serve an important function, but are orders-of-magnitude too slow to evaluate system-level performance. Real applications on parallel machines run for billions, or even trillions of cycles; even register-transfer-level simulators are much too slow.

Over the last several years, direct execution has become widely used to accelerate architectural simulations [6, 4, 3, 7, 14]. Direct execution exploits the commonality between the instruction set of the simulated *target* machine and the underlying *host* system. For example, a floating-point multiply on the target is "simulated" by executing a floating-point multiply on the host. Such a system need only simulate the differences between the target system and the host, achieving impressive performance when the two systems are very similar.

Simulations of parallel computers have exploited direct execution in several ways [3, 7, 5]. Most commonly, a parallel target system is simulated on a uniprocessor host. For example, the Tango system spawns an event generation process for each processor in a target shared-memory system. These processes directly execute all computation instructions, but must send most memory references to a central simulation process. Tango can be parallelized to a limited extent by running the event generation processes in parallel, but the central memory-system simulation process quickly becomes a bottleneck.

A recent simulator—the Wisconsin Wind Tunnel

(WWT)—extends direct execution to simulate a parallel target machine on top of parallel host (a Thinking Machines CM-5) [19]. WWT differs from earlier simulators in two ways. First, it directly executes all load and store instructions that hit in the target system's cache. Second, it integrates direct-execution with a conservative fixed-window parallel discrete-event simulation algorithm to not only parallelize event generation, but also the memory system simulation [15, 1, 17, 8, 16].

Parallel simulators like WWT are much faster than comparable uniprocessor simulators, providing the quick turn-around-time that can be so important to the design cycle. However, parallel simulation is not necessarily cost-effective for evaluating alternative parallel machines. Computer architects frequently run many independent simulations—for different applications, memory systems, and system sizes—and compare the results. Because these simulations are independent and run as a batch, parallelism can be achieved much more simply by running them simultaneously on different workstations. And since workstations have better (i.e., lower) cost/performance ratios than parallel computers, this simpler "coarse-grain" parallelism appears more cost-effective than finer-grain parallel simulators like WWT. This cost/performance differential is only exacerbated by the reality that parallel simulators rarely achieve perfect speedups.

However, in the central result of this paper, we show that parallel computer simulations are, in fact, more cost-effective than uniprocessor simulations, for sufficiently large target systems. The key intuition behind this result is that large simulations require large memory sizes, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory. Using cost models based on current commercial products and a performance model based on WWT, we show that (1) for bus-based shared-memory multiprocessors, parallel simulation becomes more cost-effective when target systems reach 16 or 32 nodes, and (2) for massively parallel systems, with their large price premium, parallel simulation becomes more cost-effective when the target system size reaches 32.

This paper also develops an analytic model of WWT's performance which incorporates three major factors: event processing time, context switch overhead, and host cache and TLB interference. We show that the variability in event processing times can be accurately modeled using Kruskal and Weiss's model for fork-join parallel programs [11]. The frequency of context switches, incurred when switching between target nodes, is accurately modeled by the maximum of binomial random variables. We extend Thiebaut and Stone's *footprint* model to predict the interference of multiple targets in the host cache and TLB [25]. Finally, we show that the model accurately estimates the measured speedup of WWT, with maximum error of 8% in three applications and 16% for all five applications.

The next section reviews the design of the Wisconsin Wind Tunnel. Section 3 develops the analytic performance model, and Section 4 compares its predictions to the measured speedups. Section 5 describes how the performance model is extended to estimate cost/performance for many target and host systems. Section 6 presents and discusses the results from this cost/performance model, and Section 7 summarizes our contributions.

2 Simulation Methodology

2.1 The Wisconsin Wind Tunnel

The Wisconsin Wind Tunnel (WWT) is a simulator for evaluating parallel computer systems—specifically cache-coherent shared-memory computers [19]. WWT uses the execution of shared-memory applications to drive a distributed discrete-event simulation of proposed hardware. Events generated by the simulation, such as cache misses and coherence messages, are used to schedule the application, permitting accurate calculation of the target system execution time.

WWT uses direct execution to exploit similarities between the target system (under study) and the host system (on which it executes). Because WWT executes on a message-passing machine (a Thinking Machines CM-5), it must simulate the shared memory abstraction using a fine-grain extension of Li's shared virtual memory [13]. Shared virtual memory uses the standard address translation hardware to control memory access on each node. When a node first accesses a shared data page, it allocates a local copy and maps it into the shared address space on that node; subsequent accesses reference the copy. Multiple read-only copies are supported using the page protection facilities. Program accesses that require a data transfer to acquire a valid or exclusive copy are signaled as page faults.

WWT's fine-grain extension uses the CM-5's error-correcting code (ECC) bits to synthesize tag bits on each 32-byte block in physical memory. Using the three tag values—*invalid*, *read-only*, and *writable*—in combination with the address translation hardware, WWT implements a distributed shared memory that maintains coherence at a finer granularity than a virtual memory page.

WWT uses logical clocks to correctly calculate the logical execution time of a target system, modeling latencies, dependencies, and queuing. WWT manages interprocessor interactions by dividing program execution into lock-step quanta (also called fixed windows [8], bounded lag [15] or time buckets [24]) to ensure all events originating on a remote node that affect a node in the current quantum are known at the quantum's beginning. WWT implements this using the CM-5's "network done" barrier, which guarantees that all messages are received before the quantum completes [12]. WWT combines this distributed simulation algorithm with direct execution by ordering all events on a node for the current quantum and directly executing the process up to its next event.

3 WWT Performance Model

One approach to evaluating cost/performance of parallel simulation is to simply measure the performance of simulation runs on several different system sizes and directly compute the cost/performance ratio. Unfortunately, while this technique provides exact results for the measured systems, it is difficult to extrapolate them to larger or smaller systems. Furthermore, simple measurements provide little or no insight into *why* a system performs as it does, making it difficult to understand the generality of the results.

In this study, we construct a performance model of the Wisconsin Wind Tunnel that accurately predicts the simulation speedup. We do this by estimating the time to simulate N target nodes on p host nodes; as we vary p, the number of target nodes per host node, $K = N/p$, changes. We then compute speedup as the ratio of the time to simulate N target nodes on a single ($p = 1$) host node over the time to simulate N target nodes on p host nodes. We calibrate the model, for each application, by extracting parameters from a small number of fully-parallel ($K = 1$) simulation runs. Section 4 discusses the accuracy of the model, and Section 5 describes how we extrapolate the model to estimate cost-performance for many different simulations.

Our model predicts the mean *running time* of a quantum. The running time of quantum i begins at the end of the $i-1$st quantum synchronization barrier and terminates at the end of ith quantum barrier, as illustrated in Figure 1. When simulating a single target node per host node ($K = 1$), two major factors dominate the quantum running time: *processing time*—the time to process simulation events including direct execution of the target program—and the *quantum overhead*—which consists of a barrier synchronization and the overhead of scheduling the target nodes. As illustrated in the left-hand side of Figure 1, the running time is simply the (fixed) quantum overhead plus the maximum processing time of any of the host nodes (e.g., host node 1 is on the critical path).

When there are multiple target nodes per host node ($K > 1$), the running time includes the sum of the processing times of each target node on the critical path. In addition, the simulation incurs both direct and indirect overheads from context switching. The direct overheads include the time to save and restore integer and floating-point registers. The indirect overheads occur because multiple target nodes compete for space in the host's cache memory and translation lookaside buffer (TLB), causing extra misses. The right-hand side of Figure 1 illustrates the case where two target nodes are simulated on each of two host nodes.

Finally, when all target nodes are simulated on a single host node ($K = N$), there can be no load imbalance so there is no waiting time. The quantum synchronization barrier becomes unnecessary and could be omitted; however, the overhead is insignificant compared to the $K = N$ processing times.

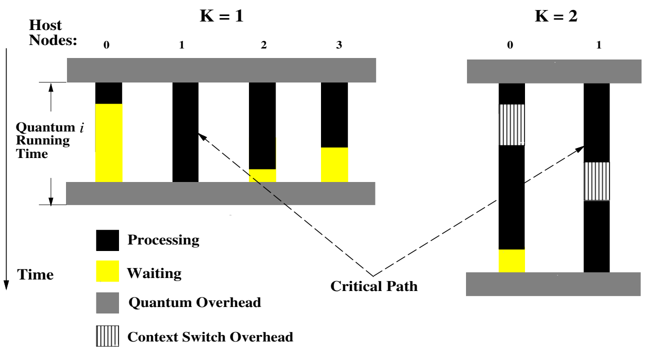

Figure 1: Running Time of a Quantum

An ideal performance model for parallel simulation would accurately predict performance simply as a function of the number of host nodes, p, and the number of target nodes per host node, K. However, such a simple model cannot account for variations between target applications, target architectures, or host systems.

To accurately model WWT's performance, we found it necessary to measure 12 parameters for each target application (and target architecture). Several of the parameters are simple event frequency measures, such as the mean number of active target nodes in a quantum. Others are timing measurements, such as the mean event processing time. Accurate, low-overhead timing measurements were made possible by the cycle counter in the CM-5's network interface unit. All parameters can be extracted from four runs of the fully-parallel simulation.

In the remainder of this section, we describe how we model each of the major contributers to simulation time: event processing time, direct context switch overhead, and host cache and TLB interference.

3.1 Modeling Processing Times

A potentially serious problem with conservative fixed-window simulation algorithms is that most host nodes will be idle while they wait for the slowest node to reach the barrier. In WWT, variations in event processing time are caused both by variation in the number of events that must be processed and in the time to process different types of events.

Within each quantum, a target node may process zero or more events. The (hopefully) common case is that a target node uses direct-execution to "simulate" local computation, including memory references that hit in its local cache. However, other events can occur, such as local cache misses and coherence messages from remote nodes. A target node may also have no events to process if, for example, the target program is waiting for a lock, barrier, or cache miss.

We have modeled this variability using a model that Kruskal and Weiss proposed for estimating the completion time of fork-join programs on MIMD parallel processors [11]. The model is asymptotically exact (as p and K go to infinity, with K growing faster than $\log p$) if the processing times are independent and identically distributed (i.i.d.) and the distribution function is increasing failure rate. However, they demonstrate that the model is remarkably robust even when these assumptions are violated.

In WWT, processing times are neither independent nor identically distributed. For example, when the target program uses barrier synchronization, target nodes that reach the barrier first will wait for the rest; since the barrier may span multiple quanta (due to target system load imbalances), the event processing times will be zero for all the waiting target nodes and hence are not independent. Moreover, a parallel program typically exhibits several distinct phases of execution, where the behavior of the program changes across phases, resulting in processing times that are not identically distributed over time. We have found that the lack of independence has little effect, perhaps since well-written parallel programs spend little time waiting for barriers, but that the phase behavior of programs is significant.

Kruskal and Weiss's model uses two parameters to characterize the workload: the mean μ and variance σ^2 of the processing times. We modify their model slightly, by using standard analysis-of-variance techniques to separate the variance within a quantum, σ^2_{inter}, from the variance of the entire population, σ^2 [10]. This modification approximates the more technically correct, but computationally expensive, alternative of computing μ and σ^2 separately for each quantum. Our model for the mean processing time in a quantum, $T_{processing}$, is simply:

$$T_{processing}(K,p) = K\mu + \sigma_{inter}\sqrt{2K\log p} \qquad (1)$$

The first term in the equation is simply the expected sum of the processing times on any host node. The second term accounts for the quantum running time being determined by the slowest host node.

3.2 Modeling Direct Overhead

Because WWT uses a separate address space for each target node, it incurs a full context switch whenever it must simulate a different target node. The direct context switch overhead includes the time to save and restore both integer and floating-point registers. Since the CM-5's SPARC processor uses register windows, the time to restore integer registers must include factors for the mean number of underflow traps per context switch (N_{win}) and the mean underflow trap service time (T_{win}).

WWT includes several optimizations to eliminate unnecessary state saving on context switches. For example, because the simulator does not use any floating-point operations, WWT only restores the floating-point registers if the target CPU needs to execute. This reduces the overhead in the case that a target only needs to process non-target CPU events, e.g., directory messages.

We approximate the mean number of context switches (N_{cxsw}) in a quantum by the mean number of target nodes scheduled for simulation. To estimate how many of these fall on the critical path, we compute the maximum of p binomial random variables, each being the sum of K flips of a coin with probability N_{cxsw}/N. Similarly, we predict the mean number of floating-point register saves and restores (N_{FP}) using a "coin" whose probability is the fraction of target nodes in a quantum that directly execute CPU events. This simple model is extremely accurate for both N_{cxsw} and N_{FP}.

Denoting the mean service time for a context switch as T_{cxsw}, and floating-point registers saves and restores as T_{FP}, our model for the direct overhead is:

$$\begin{aligned} T_{direct}(K,p) &= N_{cxsw}(K,p)\,(T_{cxsw} + N_{win}T_{win}) \\ &\quad + N_{FP}(K,p)T_{FP} \end{aligned} \qquad (2)$$

3.3 Modeling Indirect Overhead

When multiple target nodes are simulated on the same host node, they compete for space in the host's cache memory and TLB. The interference that results has a first-order effect on simulation performance. Other researchers have seen similar effects for more general parallel programs; for example, Singh, et al. recently presented significant superlinear speedups that result from cache and TLB performance improvements as the number of processors increases [21].

We use a generalization of Thiebaut and Stone's *footprint* model [25] to predict cache and TLB interference ($T_{cache\&TLB}$), as the number of target nodes per host node increases. The footprint of a process is defined to be the set of blocks that a process leaves in an infinite cache. In a finite cache, some of the blocks in the footprint will not fit, and are replaced. We define the *projec-*

Name	Input Data Set	Million Cycles
appbt	$12 \times 12 \times 12$, 15 iter	124
barnes	1024 bodies, 10 iter	95
sparse	256×256 dense	86
tomcatv	256×256, 10 iter	28
water	256 mols, 10 iter	49

Table 1: Application Programs

The table displays the application programs used in this paper. *Appbt* is a computational fluid dynamics program that solves systems of tridiagonal equations [2]. *Sparse* solves $AX = B$ in parallel for a sparse matrix A. *Tomcatv* is a parallel version of the SPEC benchmark [23]. *Barnes* and *Water* are from the SPLASH benchmarks [22].

tion of a process to be the set of blocks a process leaves in a finite cache that it may reference again[1]. Given the size of the footprints of two processes, Thiebaut and Stone's model estimates the projection of each process and uses it to determine the interference. We have extended the model to allow for sharing between processes, estimate the interference between more than two processes, and take as input the size of the projections of the processes, rather than of the footprints.

We estimate the average cache (TLB) projection of a target node by measuring the average processing time both with and without flushing the cache (TLB) at the beginning of every quantum. The difference between these times is due to refetching the blocks in the target's projection; by dividing this difference by the CM-5's cache (TLB) miss penalty, we can determine the expected size of the projection of a target node. To accurately estimate interference on the critical path, we found it necessary to not only measure the average projection, but also measure the average of the largest projection in a quantum.

3.4 Running Time of a Quantum

Putting these three submodels together, along with the fixed quantum overhead, $T_{quantumoverhead}$, allows us to estimate the mean running time of a quantum:

$$T(K,p) = T_{processing}(K,p) + T_{direct}(K,p) + T_{cache\&TLB}(K,p) + T_{quantumoverhead} \quad (3)$$

where p is the number of host processors, and $K = N/p$ is the number of target nodes per host node.

[1]Thiebaut and Stone called this the "footprint in a finite cache."

4 Validating the Model

We validate the model by simulating a 32-node cache-coherent shared-memory multiprocessor with a 4-way set-associative 32-Kbyte cache kept coherent using the Dir_1SW coherence [9, 26]. The network latency (and hence quantum length) is 100 cycles. The target system executes in one of two phases. A serial phase in which shared memory is allocated and mapped on all nodes, and a parallel phase in which a single thread of execution is initiated on every node. Since we are interested in the behavior of the simulator when all target nodes have started executing threads, we only focus on the portion of the simulation corresponding to the parallel execution of code on the target nodes.

Table 1 depicts the benchmarks used to run on the simulated system. The data sets used in this study are much smaller than one would normally run. However, in order to measure speedup, the data set had to be small enough so that we could simulate all 32 nodes of the target system on a single CM-5 node (with 32 megabytes of memory). Because the small data sets limit the available parallelism in the target programs—resulting in poor target speedups—we expect the results in this paper to be conservative. Simulations of larger data sets achieve better speedups than we observe here.

We first compare our estimate of processing time, $T_{processing}(K,p)$, against the measured sum of processing times on the critical path. The left-hand side of Figure 2 plots these times as speedup: the sum of all N processing times divided by (a) $T_{processing}(K,p)$ and (b) the measured sum of K processing times on the critical path with p host nodes. We can make two observations from these graphs. First, the model is quite accurate for *appbt*, *barnes*, and *tomcatv*, but consistently underestimates the speedup for *sparse* and *water*, with a maximum observed error of -24% for *sparse* on a 4-node host system. The model is more accurate at the extremes: it is exact, by definition, when $p = 1$, and the error is less than 16% for $p = 32$. The second, more fundamental observation, is that the inherent simulation parallelism is low, only providing speedups ranging from 4 to 9 on 32 host nodes. This is at least partially due to the low target system speedups these programs achieve for the small data sets used in this study.

Despite the relatively low "inherent" parallelism in event processing times, the Wisconsin Wind Tunnel actually achieves acceptable overall speedups, as illustrated in the right-hand side of Figure 2. These plots show the overall simulation speedups, plus a breakdown into the contributions of the various overheads. The central observation is that overhead increases the simulation parallelism by up to a factor of two. This result is consistent with additional measurements which indicate that overhead accounts for 44% to 68% of the computation in a sequential WWT simulation. These overheads

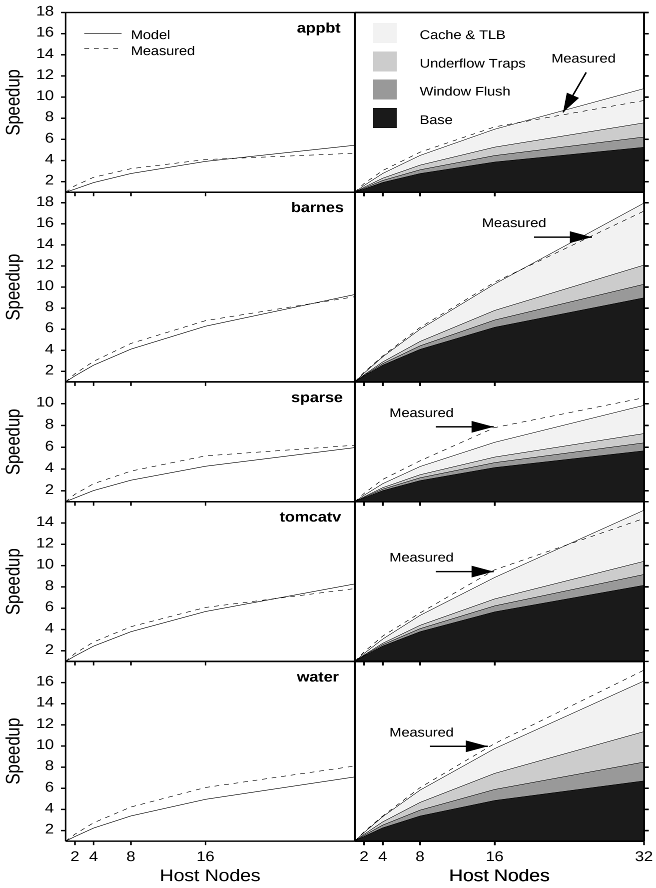

Figure 2: Speedup of Parallel Simulation

The figures plot $T_{processing}(Kp,1)/T_{processing}(K,p)$ (left) and $T(Kp,1)/T(K,p)$ (right) for $Kp = 32$ against measured speedups. The figures to right also plot the the breakdown of the major components contributing to the overall speedup. Quantum overhead is not parallelizable and therefore reduces the overall speedup. The contribution of quantum overhead and floating point registers save and restore to the speedup is not substantial and therefore is displayed in conjunction with the sum of processing times as $Base$.

not only decrease as the host nodes increase, but, to the first order, they are perfectly parallelizable. Therefore, parallel simulation benefits both from processing simulation events in parallel and distributing the overhead across multiple host nodes.

The figure also illustrates that cache and TLB interference causes significant overhead; measurements indicate that it accounts for up to 30% of the running time when $K > 4$. When more than 4 target nodes compete for the same cache and TLB, our model and measurements show that a target node will incur misses on most memory references. Parallelizing the simulation reduces the number of target nodes per host node, and hence reduces the number of cache misses on the critical path of the computation.

Although the absolute error in total speedup is roughly the same as the error due to $T_{processing}(K,p)$, the relative error is roughly half as big because overhead accounts for half the simulation time and we model overhead accurately. For four of the benchmarks the maximum error in the model is less than 12%. For *sparse* the maximum error is -16% for a 16-node host system.

5 Modeling Cost/Performance

The model introduced in Section 3 accurately predicts simulation performance for a target system with N nodes. Section 5.1 describes how we extend the model to estimate simulation performance for both larger and smaller values of N. Section 5.2 introduces our cost models for uniprocessor, bus-based shared-memory multiprocessor, and massively-parallel processor systems. Section 5.3 combines the cost and performance models to estimate cost/performance.

5.1 Scaling the Performance Model

The performance model developed in Section 3 extracts parameters from a fully-parallel ($p = N$) simulation of a specific target system, and uses them to predict the performance of that same simulation running on different numbers of host nodes. The model, however, says nothing about the simulation performance of larger or smaller target systems. To extend the model, we must make several assumptions about how the target and simulation systems scale.

We assume memory-constrained scaling [20] when we vary the size of the target system. In memory-constrained scaling, the data set size grows linearly with respect to the number of (target) nodes. This scaling model has two key properties. First, application parallelism generally increases at least linearly with data set size, so target system speedup should not limit simulation speedup. Second, this model tends to have only a minor effect on the computation/communication ratio, so that simulation processing times should have roughly the same distribution independent of N. Consequently, we can still use the mean and variance measured for a 32-node system to characterize this distribution. The Kruskal and Weiss model, used to compute $T_{processing}(K,p)$, will account for the increased (decreased) variability of larger (smaller) target system sizes.

We further assume that the overhead of multiplexing target nodes on a host node is independent of the number of host nodes, and use our earlier estimates to approximate the overhead for different target system sizes. For $K \leq 32$, we use our earlier estimates of the context switch frequency, and cache and TLB interference. We estimate the context switch frequency for $K > 32$, by linearly extrapolating the binomial model; since the tail of this curve is very close to linear, we do not expect this to introduce a significant error. We approximate the cache and TLB interference for $K > 32$, by simply using the estimated interference for $K = 32$; since both cache and TLB begin thrashing for more than 4 target nodes per host node, there will be essentially no reuse (i.e., hits) for large K.

5.2 Modeling the Cost of Host Systems

In this section, we introduce cost models for uniprocessors (Uni), small-scale bus-based shared-memory multiprocessors (Bus), and large-scale parallel supercomputers (MPP). The cost models are based on current products and allow us to vary the number of host processors, p, and the number of target nodes per host node, K. We assume that each host node requires 32 megabytes per target node. This is significantly more than needed for the small data sets used in this study; however, these data sets were chosen so that we could simulate 32 target nodes within 32 megabytes of memory (i.e., on one CM-5 node). Real data sets are much larger; for example, the official NAS input to *appbt* is 125 times larger than the data set presented here [2].

Our uniprocessor cost model is based on the Silicon Graphics CHALLENGE M, a rack-mounted uniprocessor workstation server. We use a server configuration because desktop and deskside units do not provide the necessary memory expansion capability [18]. For a target system of size K, we model the cost of a uniprocessor simulation platform as:

$$C_{Uni}(K) = BaseC_{Uni} + C_{processor} + KC_{memory} \quad (4)$$

where $BaseC_{Uni}$ denotes the base cost of the frame (box, power supply, etc.), $C_{processor}$ denotes the cost of a processor board excluding memory, and C_{memory} denotes the cost of a 32-megabyte memory module.

Bus-based shared-memory multiprocessors consist of a frame containing a variable number of processor and memory boards connected by a backplane bus. This cost model is based on the Silicon Graphics CHALLENGE XL system. The cost of a bus-based host system with p processors simulating a target system of size Kp (for all $2 \leq p \leq 40$) is:

$$C_{Bus}(K,p) = BaseC_{Bus} + pC_{processor} + KpC_{memory} \quad (5)$$

where $BaseC_{Bus}$ is the base cost for the frame.

Current implementations of massively parallel processors consist of a collection of workstation-like processing nodes connected together by a high-bandwidth interconnection network. Our cost model for these systems does not include a fixed base cost because they are generally expanded by adding entire cabinets, rather than individual processor boards. Rather than try and capture the complex step function of the actual cost, we simply approximate it as a linear function of p; this approximation should not introduce significant error since we only consider values of p that are powers of two. Modeling the network cost as a multiplier, $X_{network}$, of the processor cost, the overall cost (for all $p \geq 2$) is:

$$C_{MPP}(K,p) = p(1 + X_{network})C_{processor} + KpC_{memory} \quad (6)$$

For the purposes of this study, we use current Silicon Graphics list prices for our uniprocessor and shared-memory multiprocessor cost estimates: $C_{processor}$ = \$20000, C_{memory} = \$3200 (32 megabytes), $BaseC_{Uni}$ = \$3200, and $BaseC_{Bus}$ = \$76800 [18]. We assume $X_{network} = 2$, which is a reasonable estimate of network cost for current generation MPP systems. Ultimately, we expect competition to reduce $X_{network}$ to values of $0.1 \sim 0.5$.

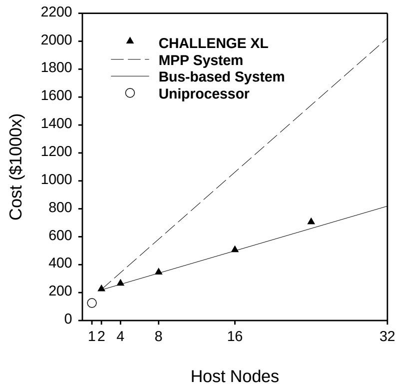

Figure 3: Modeling the Cost of Host Systems

Figure 3 plots the cost models as a function of the number of host nodes for a 32-node target system. The minimum cost of a parallel host is approximately two times the cost of a uniprocessor host system. The figure also depicts the prices of Silicon Graphics CHALLENGE XL bus-based multiprocessor servers [18]. The cost curve for the massively parallel processors has a much steeper slope as compared to the curve of the bus-based multiprocessors due to the high cost of the interconnection network per node.

5.3 Modeling Cost/Performance

Since speedup is a measure of parallel simulation performance, cost/performance is simply the cost of the host system divided by the simulation speedup it achieves. For a uniprocessor system, the cost/performance is simply C_{Uni}, because speedup is 1 by definition. For parallel simulation of a Kp-node target system, the cost/performance is:

$$CP_{Machine}(K,p) = \frac{T(K,p)}{T(Kp,1)} C_{Machine}(K,p) \quad (7)$$

where $Machine$ is either Bus or MPP.

6 Analyzing Cost/Performance

Given a model for parallel simulation cost/performance, there are two questions that we would like to address. First, is parallel simulation simply faster than sequential simulation, or is it also more cost-effective? Second, if we have parallel simulation, what value of K achieves the best cost/performance?

We address the second question first, by analyzing the asymptotic behavior of K_{min} using a simplified form of the cost/performance model. We know that such a minimum value exists, because the cost function is increasing linear in p and the speedup is a bounded convex function in p. Therefore, for a fixed target system size, the cost/performance function is concave with a minimum at $K = K_{min}$. We simplify the model slightly to clarify the asymptotic analysis. We approximate the running time of a quantum as $aK + b\sqrt{K \log p}$, where the first term accounts for factors contributing linearly to the running time such as the mean processing time of target nodes and the per-target node overhead on the critical path, and the second term accounts for the variation in the sum of processing times on the critical path. We also approximate the cost function as $p(C_{processor} + KC_{memory})$. The cost/performance function will then be:

$$CP_{asymptotic}(K,p) = C_{memory}(K + \frac{C_{processor}}{C_{memory}}) (1 + \frac{b}{a}\sqrt{\log p}\frac{1}{\sqrt{K}}) \quad (8)$$

For a given target system size (i.e., fixed N), the above function has a minimum at K_{min} which is an increasing function of $C_{processor}/C_{memory}$, b/a and $\sqrt{\log p}$. Since the variation of the latter is negligible in the range of feasible values of p, a key contribution of this model is that K_{min} is, to the first order, independent of p.

The term b/a is reciprocally proportional to the amount of parallelism available in the simulation. Small values of b/a can result from either (a) processing times that have a small coefficient of variation, and thus cause little load imbalance, or (b) small mean processing times which cause the—perfectly parallelizable—overhead to dominate. In either case, high parallelism results in small values of K_{min}. This result is intuitive, since higher parallelism gives rise to larger speedups which in turn offset the cost of adding more host nodes.

The model also predicts that a decrease in the cost of memory with respect to the cost of a processor board results in a larger value of K_{min}. The intuition behind this result is that, for a given target system size, decreasing memory cost increases relative processor cost, and shifts the balance toward more memory intensive simulations.

Unfortunately, analyzing the simplified model does not help us answer the first question of when is parallel simulation better than sequential simulation. Instead, we graphically examine the full model. The upper half of Figure 4 plots $CP_{Bus}(K,p)$ and $C_{Uni}(Kp)$ for the simulation of *appbt* and *barnes*. These two applications are reasonably representative of the 5 benchmarks: *appbt* and *sparse* exhibit relative low speedups, while *barnes*, *tomcatv*, and *water* exhibit considerably higher speedups. Figure 4 shows that although uniprocessor simulation is more cost-effective for small target systems, up to 8 or 16 nodes, parallel simulation offers superior cost/performance as the target system grows beyond approximately 16 nodes. More important than the exact numbers, the trend clearly shows that parallel simulation becomes increasingly cost-effective as the target system grows.

The second interesting prediction of this model is the lack of continuity in p. That is, parallel simulation does not gradually become more effective, but rather once the speedup is sufficient to overcome the large base cost, the optimum cost/performance occurs when the simulation is either fully parallel (*barnes*) or nearly so (*appbt*).

The lower half of Figure 4 plots $CP_{MPP}(K,p)$ and $C_{Uni}(Kp)$ for *appbt* and *barnes* and $Kp \leq 128$. The figure illustrates that uniprocessor simulation is more cost-effective than parallel simulation for target systems of up to 16 nodes. The large jump in cost/performance as p increases from 1 to 2 nodes is due to the significant premium charged by MPP vendors. For host systems of up to 16 nodes, the simulation speedup is not large enough to offset this premium and therefore uniprocessor simulation offers better cost/performance. Minimum cost/performance for larger systems lies consistently at 4 and 8 target nodes per host node for the two benchmarks independent of the number of host nodes. Moreover, the larger parallelism available in the simulation of *barnes* results in optimum cost/performance at a smaller value of K. These results are in accord with the predictions of our model for the asymptotic behavior of K_{min}.

Decreasing memory cost not only shifts K_{min} towards larger values—confirming our analysis of the simplified model—but increases the target system size at which parallel simulation becomes more cost-effective than sequential simulation. For example, decreasing the memory cost (or, equivalently, the simulation's memory requirement) by a factor of four increases the break-even point for parallel simulation to 128 target nodes.

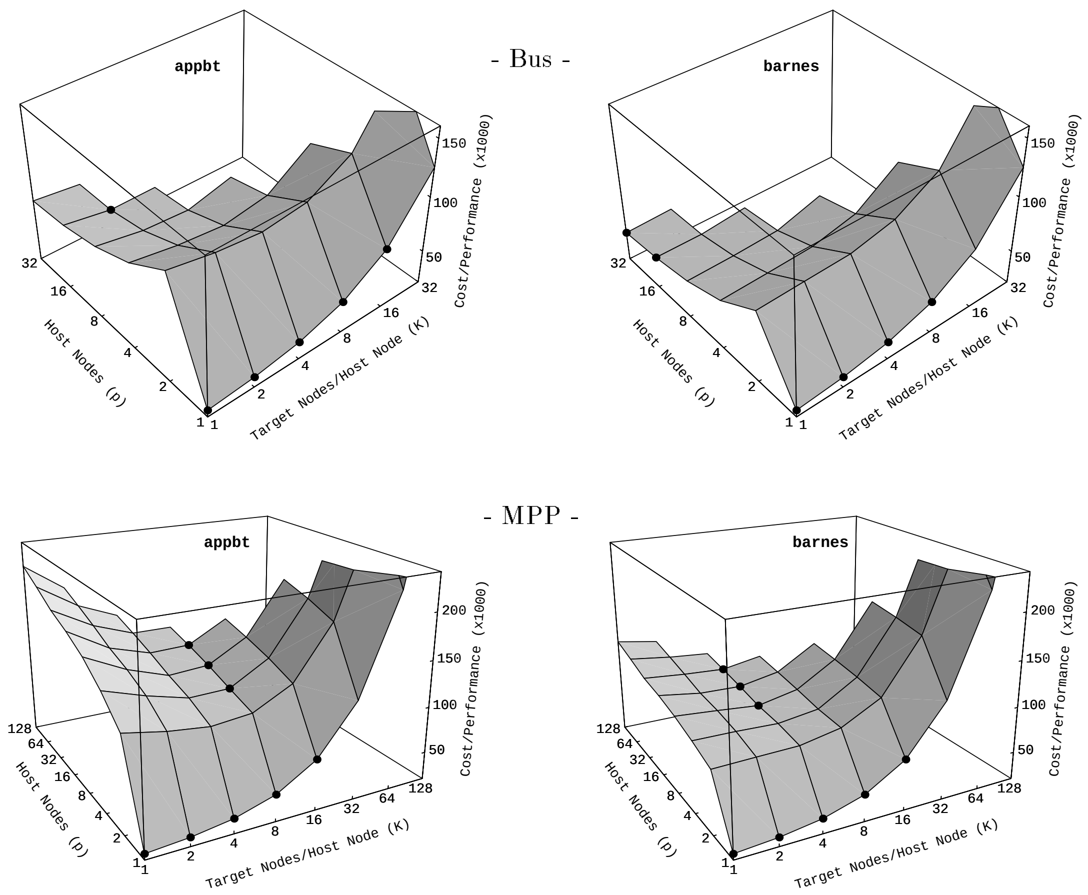

Figure 4: Cost/Performance of Parallel Simulation

The figures plot $CP_{Bus}(K,p)$ (top) and $CP_{MPP}(K,p)$ (bottom) against $C_{Uni}(Kp)$ for two benchmarks. *appbt* and *barnes* are representatives of classes of parallel simulation which exhibit low and high speedups respectively. The cost parameters are the same as those in figure Figure 3. A bullet at coordinates (K,p) indicates the minimum cost/performance for the simulation of a target system of size Kp.

Decreasing the processor cost (and/or the cost of the network for MPP's) has a complementary effect, not only decreasing K_{min}, but reducing the break-even target system size. Similarly, increasing the parallel simulation speedups, as we expect for larger data sets, will also tend to make parallel simulation increasingly cost-effective.

7 Summary and Conclusions

This paper examines the cost/performance of simulating a hypothetical *target* parallel computer using a commercial *host* parallel computer. We address the fundamental question of whether parallel simulation is simply faster than sequential simulation, or whether it is also more cost-effective. We answer this by developing a performance model of the Wisconsin Wind Tunnel (WWT) that incorporates three major factors: event processing time, context switch overhead, and host cache and TLB interference. For the performance model, we show that:

- the variability in event processing times can be accurately modeled using Kruskal and Weiss's model for fork-join parallel programs;
- the frequency of context switches, incurred when switching between target nodes, is accurately modeled by the maximum of binomial random variables;
- an extension of Thiebaut and Stone's *footprint* model accurately predicts the interference of multiple targets in the host cache and TLB;
- the performance model's predictions of simulation speedup are within 10% on average and are always within 20% for these workloads.

We then combine the performance model with simple cost models and show—in the central result of this paper—that parallel computer simulations are, in fact, more cost-effective than uniprocessor simulations. The key intuition behind this result is that large simulations require large memory sizes, which dominate the cost of a uniprocessor; parallel computers allow multiple processors to simultaneously access this large memory. Furthermore, we show that K_{min}, the number of target nodes simulated per host node for optimum cost/performance, is essentially independent of p, the number of host processors. Using cost models based on current commercial products and a performance model based on WWT, we show that (1) for bus-based shared-memory multiprocessors, parallel simulation becomes more cost-effective when target systems reach 16 or 32 nodes, and (2) for massively parallel systems, with their large price premium, parallel simulation becomes more cost-effective when the target system size reaches 32.

8 Acknowledgements

This work is part of the Wisconsin Wind Tunnel project, which is co-led by Profs. Mark Hill, James Larus, and David Wood and funded by the National Science Foundation. We would like to especially thank Steve Reinhardt for all his contributions to developing the Wisconsin Wind Tunnel. We would also like to thank Alain Kägi for his helpful comments that helped improve this work.

References

[1] Rassul Ayani. A Parallel Simulation Scheme Based on the Distance Between Objects. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 113–118, March 1989.

[2] David Bailey, John Barton, Thomas Lasinski, and Horst Simon. The NAS Parallel Benchmarks. Technical Report RNR-91-002 Revision 2, Ames Research Center, August 1991.

[3] Bob Boothe. Fast Accurate Simulation of Large Shared Memory Multiprocessors. Technical Report CSD 92/682, Computer Science Division (EECS), University of California at Berkeley, January 1992.

[4] Eric A. Brewer, Chrysanthos N Dellarocas, Adrian Colbrook, and William Weihl. PROTEUS: A High-Performance Parallel-Architecture Simulator. Technical Report MIT/LCS/TR-516, MIT Laboratory for Computer Science, September 1991.

[5] Robert F. Cmelik and David Keppel. Shade: A Fast Instruction-Set Simulator for Execution Profiling. Technical Report UWCSE 93-06-06, Department of Computer Science, University of Washington, 1993.

[6] R.C. Covington, S. Madala, V. Mehta, J.R. Jump, and J.B. Sinclair. The Rice Parallel Processing Testbed. In *Proceedings of the 1988 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 4–11, May 1988.

[7] Helen Davis, Stephen R. Goldschmidt, and John Hennessy. Multiprocessor Simulation and Tracing Using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (Vol. II Software)*, pages II99–107, August 1991.

[8] Richard M. Fujimoto. Parallel Discrete Event Simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[9] Mark D. Hill, James R. Larus, Steven K. Reinhardt, and David A. Wood. Cooperative Shared Memory: Software and Hardware for Scalable Multiprocessors. *ACM Transactions on Computer Systems*, 11(4):300–318, November 1993. Earlier version appeared in ASPLOS V, Oct. 1992.

[10] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, 1991.

[11] C. P. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Trans. on Software Engineering*, 11(10):1001–1016, October 1985.

[12] Charles E. Leiserson, Zahi S. Abuhamdeh, David C. Douglas, Carl R. Feynman, Mahesh N. Ganmukhi, Jeffrey V. Hill, W. Daniel Hillis, Bradley C. Kuszmaul, Margaret A. St. Pierre, David S. Wells, Monica C. Wong, Shaw-Wen Yang, and Robert Zak. The Network Architecture of the Connection Machine CM-5. In *Proceedings of the Fifth ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, July 1992.

[13] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):321–359, November 1989.

[14] Jia-Jen Lin. *Efficient Parallel Simulation for Designing Multiprocessor System.* PhD thesis, University of Michigan, Ann Arbor, 1992.

[15] Boris D. Lubachevsky. Efficient Distributed Event-Driven Simulations of Multiple-Loop Networks. *Communications of the ACM*, 32(2):111–123, January 1989.

[16] Jayadev Misra. Distributed-Discrete Event Simulation. *ACM Computing Surveys*, 18(1):39–65, March 1986.

[17] David Nicol. Conservative Parallel Simulation of Priority Class Queueing Networks. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):398–412, May 1992.

[18] Ed Reidenbach. *CHALLENGE Server Perdiodic Table.* Silicon Graphics Computer Systems, October 1993.

[19] Steven K. Reinhardt, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, and David A. Wood. The Wisconsin Wind Tunnel: Virtual Prototyping of Parallel Computers. In *Proceedings of the 1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, May 1993.

[20] Jaswinder Pal Singh, John L. Hennessy, and Anoop Gupta. Scaling Parallel Programs for Multiprocessors: Methodology and Examples. *IEEE Computer*, 26(7):42–50, July 1993.

[21] Jaswinder Pal Singh, Truman Joe, Anoop Gupta, and John L. Hennessy. An Empirical Comparison of the Kendall Square Research KSR-1 and Stanford DASH Multiprocessor. In *Proceedings of Supercomputing 93*, pages 214–225, November 1993.

[22] Jaswinder Pal Singh, Wolf-Dietrich Weber, and Anoop Gupta. SPLASH: Stanford Parallel Applications for Shared Memory. *Computer Architecture News*, 20(1):5–44, March 1992.

[23] SPEC. SPEC Benchmark Suite Release 1.0, Winter 1990.

[24] Jeff. S. Steinman. SPEEDES: A Multiple-Synchronization Environment for Parallel Discrete-Event Simulation. *International Journal in Computer Simulation*, 2:251–286, 1992.

[25] D. Thiebaut and H.S. Stone. Footprints in the cache. *ACM Transactions on Computer Systems*, 5(4):305–329, November 1987.

[26] David A. Wood, Satish Chandra, Babak Falsafi, Mark D. Hill, James R. Larus, Alvin R. Lebeck, James C. Lewis, Shubhendu S. Mukherjee, Subbarao Palacharla, and Steven K. Reinhardt. Mechanisms for Cooperative Shared Memory. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 156–168, May 1993.