Developing Parallel Applications Using High-Performance Simulation

Eric A. Brewer
William E. Weihl
MIT Laboratory for Computer Science*

Abstract

Researchers already use high-performance simulators for algorithm development and architectural studies. Advances in simulation technology and workstation performance have made program development on top of simulators — debugging, testing, and some tuning — fast enough for real applications. Simulators provide many advantages over running directly on a multiprocessor, including versatility, trivial repeatability, and detailed nonintrusive data collection and debugging. We make the case for application development via simulation and address simulation's traditional disadvantages. We also propose a development methodology that integrates simulation into the multiprocessor development environment. Finally, we examine the software engineering issues required by this integration and the use of parallel simulators for development.

We believe that most multiprocessor application development should be done on top of a high-performance simulator (running on a workstation) rather than directly on the multiprocessor. Researchers already use these high-performance simulators for algorithm development [CBDW91], architectural studies [HM92], and language and compiler design [WBC$^+$91, HWW93]. The primary advantages of development via simulation are the cost effectiveness of workstations, the ability to exploit powerful sequential debuggers, the support for nonintrusive data collection and invariant checking, and the versatility of simulation.

Although traditional multiprocessor simulators are too slow to run real applications, high-performance simulators, including TangoLite [DGH91], RPPT [CDJ$^+$91], and our own system PROTEUS [BDCW91], achieve orders of magnitude speedup over traditional simulators; typical development simulations take only a few minutes. The rapid increases in workstation performance will continue to diminish the turn-around time.

Although we designed PROTEUS to explore algorithms and language issues, our experience and that of other users indicates that development is easier and faster with PROTEUS than with real multiprocessors. For example, Colbrook found that the development of novel search-tree algorithms was about six times faster using PROTEUS compared to the previous development of search-tree algorithms on a transputer-based multiprocessor [Col, CBDW91], even though the algorithms developed on PROTEUS were far more complex. Furthermore, a typical simulation took only about a minute and produced higher quality measurements. Although our data comes from PROTEUS, the methodology applies to high-performance simulation in general.

After discussing high-performance simulators in Section 1, we make the case for application development via simulation. We discuss the advantages and disadvantages of development on simulators in Sections 2 and 3, and then propose a methodology in Section 4. We change gears starting with Section 5, where we examine the issues that arise given this approach to development. These issues include simulating from a log, extending sequential debuggers, and ensuring source-code compatibility between the simulator and the target multiprocessor. Finally, in Sections 6 and 7, we address development on a concurrent simulator and summarize our conclusions.

1 Background

Traditional multiprocessor simulators interpret one instruction at a time. This allows arbitrary accuracy at tremendous cost. Such simulators typically require around 200 instructions per simulated instruction [CLN90], although the fastest versions (with less accuracy) can achieve 20–40 instructions per simulated instruction for uniprocessor architectures [Bed90].

High-performance simulators like PROTEUS and TangoLite use *direct execution* to improve performance. Most simulated instructions are mapped to host instructions and thus approach zero overhead. Typically, assembly language basic blocks must be augmented with code that tracks the passage of simulated time, so these simulators have a lower bound of about two instructions per simulated instruction in practice. Early uses of direct execution for simulation include work by Fujimoto [FC88],

*This appeared in the 1993 Workshop on Parallel and Distributed Debuigging, San Diego, California. E-mail: `brewer@lcs.mit.edu` This work is supported by the National Science Foundation, grant CCR-8716884; by DARPA, Contract N00014-91-J-1698; by an equipment grant from Digital Equipment Corporation; and by grants from AT&T and IBM. Eric Brewer is supported by an Office of Naval Research Fellowship. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. government.

Mathieson and Francis [MF88], and Covington *et al.* [CMM+88, CDJ+91].

Instructions that cannot be mapped to host instructions are simulated essentially via procedure calls. Such instructions are called *nonlocal instructions*, and include accesses to shared memory, network instructions, and in general, all instructions that may directly affect other nodes. The relative percentage of such instructions determines the performance of the simulator; for example, message-passing applications typically have many fewer nonlocal instructions and thus better simulation performance than shared-memory applications [DGH91].

2 Advantages

The advantages of simulation for development stem from three sources: running on a standard workstation, running as a single process, and running on a versatile layer of software rather than directly on real hardware.

2.1 Workstation Advantages

There are three primary advantages to running on a workstation:

1) Workstations are cost effective.
Multiprocessor cycles are fundamentally more expensive than workstation cycles. The huge amount of memory, the high bandwidth, the low volume, and the administrative overhead all make supercomputers more expensive. Large multiprocessors are cost-effective for applications that require the memory and bandwidth; however, this class generally excludes applications under development.

In addition, many multiprocessor users must pay directly for every cycle, which greatly increases the marginal cost of cycles compared to workstations. For these users, development via simulation is substantially more cost effective.

2) Multiprocessors are a scarce resource.
Running debugging code on multiprocessors wastes cycles that could be spent on productive code. Moving most development to workstations would free up these cycles for more productive runs. This is true even though most development is done on small partitions.

3) Workstations are always available.
Unlike supercomputer cycles, workstation cycles may be used anytime, without any administration. For example, interactive CM-5 [TMC92] use at the NCSA is limited to 15 minutes at a time, and batch jobs incur significant latency. The University of California at Berkeley has a similar policy for their CM-5; in practice, interactive use is painful enough that debugging amounts to instrumenting code with print statements [Yel92]. In contrast, turn-around times for PROTEUS are typically less than two minutes, and debugging can be done interactively.

2.2 Single-Process Advantages

A second set of advantages comes from running sequentially in one address space:

1) The simulation is repeatable.
Repeatability is standard for sequential applications. The traditional debugging methodology of iteratively tracking down a bug is based on repeatability. Although software logging and replay techniques can provide repeatability on multiprocessors (with some alteration of behavior) [LMC86, NM92], repeatability on simulators is trivial, nonintrusive, and always available. Section 5.1 discusses logging and replay in more detail.

2) Users can exploit sequential debuggers.
The best sequential debuggers remain far ahead of the those available on multiprocessors. We regularly run PROTEUS on top of `dbx` [Lin90] and GNU's `gdb` [SP92]. PROTEUS includes a sophisticated internal debugger [BD92] that complements the sequential debugger with features for the parallel aspects of the application (see Section 5.2).

For example, to single step the source code of a particular thread, one would use the PROTEUS debugger to reach the target thread at the approximate time, and then switch to the sequential debugger to single step the thread. The sequential debugger behaves essentially the same as it would for a true sequential program.

3) Users have access to all state.
Running in a single address space provides instant access to all of the parallel program's state. This allows simple checking of *global* invariants, and simplifies debugging. Checking a global invariant on a distributed-memory multiprocessor is *far* more difficult, and is difficult even on a shared-memory machine, since ideally all of the threads should stop during the check. Furthermore, global invariants can be checked without perturbing the behavior of the application, as discussed in Section 2.3. For example, PROTEUS nonintrusively verifies the internal state of (concurrent) global memory allocation, much like debugging versions of `malloc` in sequential C.

The ability to access global state quickly and easily simplifies debugging. For example, in the development of a parallel version of PROTEUS (discussed in Section 6), we have implemented a highly concurrent priority queue that forms the primary data structure for the parallel discrete-event simulation engine. The instantaneous access to global state allows us to produce coherent snapshots of the data structure and to verify such snapshots against desired invariants. This kind of run-time check is very difficult on a real multiprocessor.

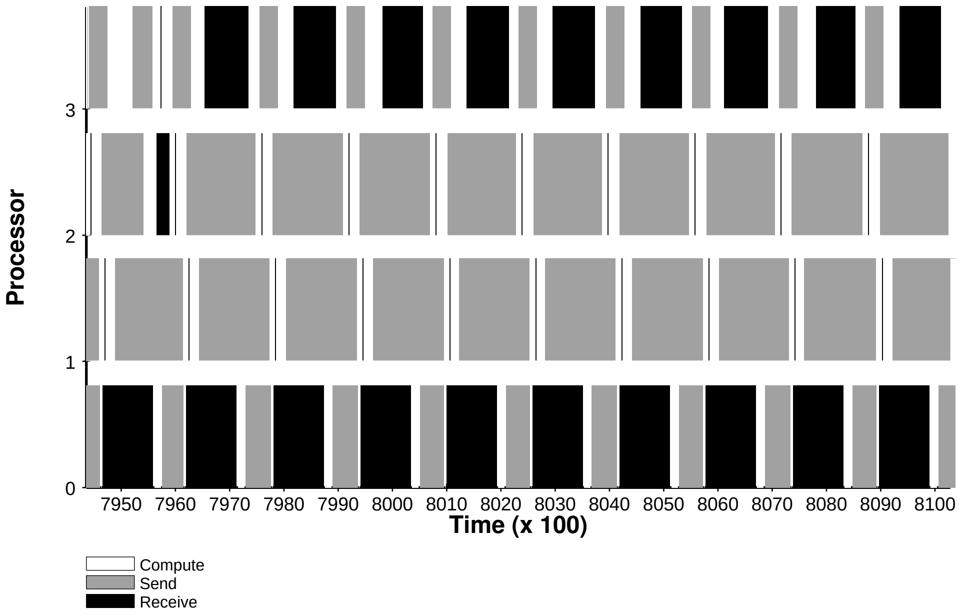

Figure 1: This graph shows the state of four stages in a pipeline search tree application [CBDW91]. The state graph is generally periodic; the width of the period reveals the throughput of the pipeline. The stages progress upward, so a single operation has a "slope" of about 60 degrees from the positive time axis: the pipeline latency can be measured directly from this slope. More importantly, this graph reveals that this particular algorithm is spending all of its time performing communication: very little of the graph is white. A change to buffered asynchronous message passing resolved this problem.

2.3 Simulation Advantages

The third set of advantages comes from running on simulated hardware rather than directly on the host architecture:

1) Debugging can be nonintrusive.
On real multiprocessors, debugging and monitoring code interferes with the application code. Even simple logging code for replay can alter behavior [Gai86]. This effect can be totally eliminated when running on top of a simulator, since it is possible to assign arbitrary code a cost of zero simulated cycles [BDCW91, CJS89]. When combined with repeatability, nonintrusive debugging allows users to add extra debugging code without concern that the symptoms will disappear due to nondeterminism. Replay systems can provide this to a lesser extent [LMC86].

2) Simulators can measure everything.
Unlike real multiprocessors, a simulator like PROTEUS can measure any aspect of the application. For example, network contention, one-way message latency, and cache behavior are all difficult to measure on a real machine, but easy with PROTEUS. Furthermore, PROTEUS provides tools that allow users to measure almost any aspect of their application nonintrusively.

One very useful graph that is difficult or impossible to obtain on real multiprocessors is a detailed profile of each processor including synchronization and OS operations. A similar graph plots the "state" of each processor versus time; this graph implies the percentage of time spent in each state and also reveals transient and correlated behavior. These graphs, called "state graphs", have proved very useful for debugging concurrent data structures; Figure 1 shows a state graph for a concurrent search tree [CBDW91].

3) Simulators are versatile.
Simulators are not limited to one target, nor even to existing machines. In fact, high-performance simulators can be used to develop applications for a machine before it is completed (or before users can acquire access). For example, we started developing large applications for the CM-5 on top of PROTEUS months before we received our CM-5. (We still develop on PROTEUS even though we now have the CM-5.) Similarly, DEC used high-performance simulation to develop software for Alpha workstations before they were available [DBMI88]; when the Alpha prototypes arrived, most of the code worked correctly the first time.

We also exploit this versatility to study portability. Applications written in the research language Prelude

[WBC+91] run on a wide variety of simulated machines, including both message-passing and shared-memory multiprocessors. Switching the target architecture requires only recompilation using the parameters of the new architecture.

3 Putative Disadvantages

There are five disadvantages to using simulators that have kept them from being a realistic option in the past. However, all of the fundamental problems have been overcome. The primary disadvantages of simulation are:

1. Simulators are too slow,
2. Simulators are too inaccurate,
3. Simulators execute only one of many non-deterministic orderings,
4. Simulators require small data sets, and
5. Simulators are not source-code compatible with the target multiprocessor.

The following subsections address these issues with the intent to show that no fundamental obstacles remain. That is, the remaining obstacles require some software engineering, but no new technology.

3.1 Simulators are Too Slow

This has always been the biggest disadvantage of simulation. However, simulators like PROTEUS and TangoLite can run real applications on simulated machines with 64 processors in just a few minutes. In developing concurrent search algorithms, Colbrook [CBDW91] found that the vast majority of his simulations took less than two minutes; he ran many thousands of simulations to test and tune the algorithms on a variety of architectures. Similarly, in studying software support for multiple contexts [WW93], Waldspurger regularly ran several *thousand* simulations over a weekend, averaging roughly a minute per simulation. Finally, many large applications have been developed on high-performance simulators, including most of the SPLASH benchmark programs [SWG92], the Prelude run-time system [WBC+91], and a parallel version of PROTEUS.

Table 1 lists several medium-size applications and their simulation performance on PROTEUS on a DECstation 5000/200. Message passing is similar to active messages [vCGS92]; shared-memory architectures use a directory-based cache-coherence algorithm [Cha90]. The MIPS column is the effective speed of one simulated processor; it is the total number of cycles simulated (for all processors) divided by the wall-clock time. The cycles/second column divides the MIPS value by the number of simulated processors to determine the rate at which the entire machine progresses.

SOR PDE Solver: This solves elliptical PDE's using successive overrelaxation. It is written in C and uses active messages and fine-grain synchronization for communication.

Counting Network [HWW93]**:** This application is written in PRELUDE [WBC+91] and has a very high communication to computation ratio. It was compiled for both message-passing and shared-memory machines. The shared-memory version performs poorly because of an extremely large amount of write sharing, which requires many cache-coherence protocol messages.

Water, SPLASH Benchmark [SWG92]**:** This program simulates water molecules and is quite taxing on shared memory. Production runs might involve 1 million molecules, but 32 is sufficient for development; 512 molecules is enough for some tuning, but given the simulation time, it would be better to tune this application on the multiprocessor.

Concurrent B-Tree [HWW93]**:** This highly concurrent B-tree algorithm is written in PRELUDE, and produces 34,000 lines of C code, including the run-time system. The program creates a B-tree with 10,000 keys and then 16 processors pump 800 operations through the tree. The very high simulation performance is due in part to the idle time at the 16 processors; simulating idle cycles takes virtually zero real time.

8-Queens: This application recursively places queens on one row and then spawns remote threads on random processors to handle the next row for each successful placement. It uses both message passing and shared memory.

Although insufficient for production runs, simulators such as PROTEUS can clearly handle development-size test cases. Additionally, the speed of workstations will continue to increase, probably *much* faster than the size of test cases. The simulator is *not* intended for huge data sets; final testing and tuning involving production-size runs should be done on the real machine.

3.2 Simulators are Too Inaccurate

Despite its high performance, PROTEUS can accurately simulate actual multiprocessors. In particular, it provides sufficient accuracy for nearly all debugging and testing and for some tuning as well. One validation study [BDCW91] reproduced published data for sorting algorithms [Qui89] on an nCUBE. After we set the communication costs according to independent data (but *before* we ran the code), all data points were within 4% of the

Simulated Application	Simulated Topology	Wall Clock	MIPS	Machine Cyc/Sec
SOR PDE solver, 1024x1024 grid, 10 iterations	8x8 Torus	1.2 min	2.7	42K
SOR PDE solver, 1024x1024 grid, 50 iterations	8x8 Torus	5.0 min	3.2	50K
Counting Network, message passing, 24 nodes, 8192 ops.	5x5x5 Mesh	22 sec	3.0	24K
Counting Network, shared memory, 24 nodes, 8192 ops.	5x5x5 Mesh	26 min	0.041	330
Water (SPLASH), shared memory, 32 molecules	4x4 Mesh	2.2 min	≈ 5	$\approx$ 300K
Water (SPLASH), shared memory, 512 particles	8x8 Mesh	40 min	≈ 4	$\approx$ 60K
Concurrent B-Tree, message passing, 10,800 operations	8x8 Mesh	2.2 min	42	656K
Concurrent B-Tree, shared memory, 10,800 operations	8x8 Mesh	18 min	3.9	61K
8-Queens, shared memory (all solutions)	2^6 Hypercube	10 sec	5.0	78K

Table 1: Performance of typical applications.

actual value. Thus, this application could have been developed entirely on the simulator.[1] Similar validations were performed for the RPPT simulator [CDJ+91]; these include FFT, LU decomposition, Gaussian elimination, and eigenvector applications running on a 16-processor iPSC hypercube or a 20-processor Sequent Symmetry. Most data points were within 3% of the actual execution time; the worst error was about 8%. Section 3.2.2 presents one validation study for PROTEUS in more detail. First, however, it is useful to examine the sources of inaccuracy in high-performance simulators.

3.2.1 Sources of Inaccuracy

There are several sources of inaccuracy in simulators that use direct execution. A primary one is the difference in instruction sets between the host and target. This has not been a problem in practice, primarily because the instruction sets for RISC processors are quite similar. For CISC instruction sets, it has been sufficient to fold a uniform scaling constant (determined empirically) into the cost of basic blocks [BDCW91]. The intuitive reason for the success of this simple approach is that so many instructions are executed that it suffices to get the average right.

There are techniques to remove this source of inaccuracy. In particular, *cross-profiling* [CJS89] compiles the high-level source into assembly for both the target and the host and sets the cost of a block on the host to match the cost of the corresponding block on the target. This eliminates errors due to the differences in instruction sets.

A simpler way to eliminate this source of inaccuracy is to use a workstation with the same processor as the target multiprocessor, such as a SPARC [SPA92] workstation for simulating a CM-5.

A second source of inaccuracy is the requirement that the cost of a block be computed statically. This may be difficult to do perfectly for superscalar processors, since the path to the block affects its cost, although the error is at most 5–10% [Lar93]. The QP profiler [BL92] computes costs based on paths, not nodes, and can thus very accurately predict the cost of the block. Members of the Environment for Portable Parallel Programming project (EPPP) [VSB+92] in Canada have extended PROTEUS with path-based profiling to accurately simulate multiprocessors based on superscalar SPARC processors.

Another source of inaccuracy is the vector units that are often available on multiprocessing nodes. Although vector units can be modeled fairly easily, the models must reflect the ability of the compiler to exploit the vector unit, which varies quite a bit for different compilers. Integrating support for simulation into the compiler removes this problem, since the simulator then knows the actual scheduling of the vector operations. In general, vector units have received very little attention in high-performance simulators. This is in part because inaccurate vector models limit only the ability to tune the application, since the functionality of the vector units is simulated exactly.

Finally, the multiprocessor operating system may not be accurately reproduced in the simulation. For example, the simulation ignores network contention due to other users and delays due to time slicing. These delays could be modeled for a specific operating system, but no attempt to do so has been made to our knowledge. Many important OS costs, such as the OS overhead for sending a message, are included in the simulation and can often be determined statistically (as was done for the validation of PROTEUS against the nCUBE).

3.2.2 Evidence for Accuracy

A second validation against a real multiprocessor appears in Figure 2. In this study, code that compares several locking algorithms on the Sequent Symmetry and the BBN Butterfly [MCS91] was rerun on PROTEUS.

The PROTEUS results correlate extremely well with the published data. It should be pointed out that the PROTEUS

[1] The communication cost parameters were derived from independent data provided by David Culler of the University of California at Berkeley.

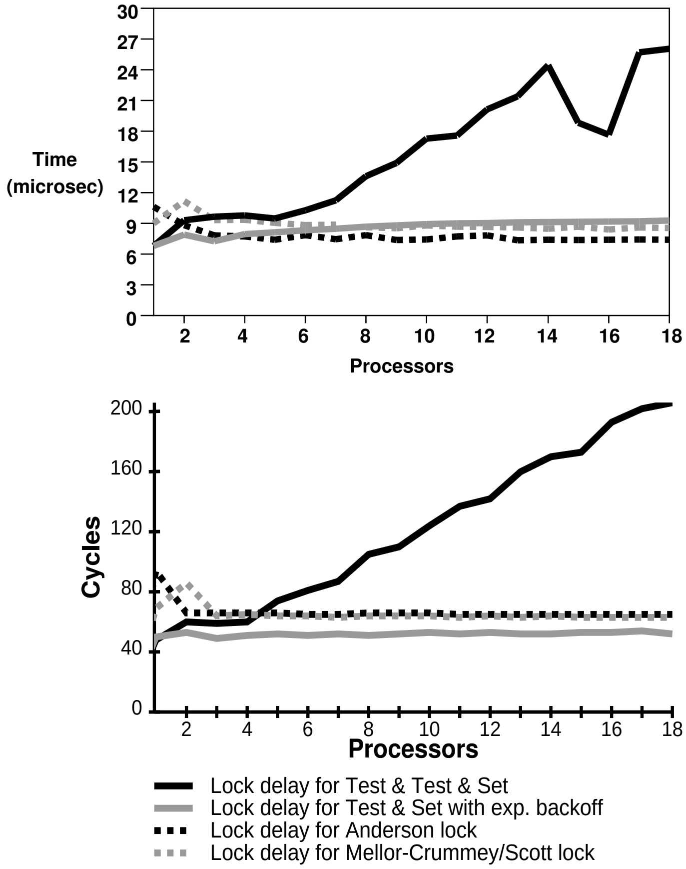

Figure 2: The top graph is published data for the Sequent Symmetry [MCS91]. The bottom graph is the same code run on PROTEUS. With the exception of the (counterintuitive) dip in the top graph, which is discussed in Section 3.2.2, PROTEUS accurately reproduces the published data. Note that these algorithms involve a lot of contention and shared-memory race conditions, which make them a formidable challenge for a simulator.

graph shown here is the *first* graph produced for these algorithms; that is, we did not modify the parameters until the graphs matched. PROTEUS correctly predicts nearly all of the individual data points, all of the slopes, and the effects for small numbers of processors.

The only discrepancy is the dip that appears in the Sequent graph but not in the PROTEUS graph. This dip implies that if you have 14 processors spinning on a lock and you *add* two more, then the average waiting time for lock acquisition goes *down*. This is highly counterintuitive and Mellor-Crummey and Scott were unable to explain the dip, although it was repeatable. In fact, the behavior predicted by PROTEUS is the expected behavior; i.e., the slope continues as before. Excluding the dip, the slope of the Test & Test & Set algorithm matches extremely well. The other algorithms all match quite well, except their relative order has changed for large numbers of processors; this is probably due to differences in the compiler since all the points are within 15%.

Thus, despite the various sources of inaccuracy and the contention and race conditions, this application could have been developed *and tuned* on the simulator. We expect that the vast majority of applications can be debugged and tested effectively via simulation; a subset of those can also be tuned to various degrees. In the end, some tuning and debugging may still be required on the multiprocessor, but more accurate simulations reduce the need for multiprocessor debugging and tuning.

3.3 Simulators Test One Ordering

Because simulators are deterministic they traditionally explore only one possible execution of a nondeterministic program; there may be latent bugs in other legal executions. PROTEUS addresses this problem by choosing between events with the same timestamp pseudo-randomly. Thus, the program is deterministic for a particular seed, but different orderings occur with different seeds. More unusual orderings may be selected by inserting random delays, although this seems to reveal errors only rarely.

Multiprocessors share this behavior to a lesser extent; often only a small percentage of legal orderings are actually executed. In particular, running on a multiprocessor provides no guarantee that all legal orderings are bug free. For both simulators and multiprocessors, tools that either statically detect race conditions [NDD93] or systematically test many or all orderings [TLK92] are desirable.

Thus, although simulators may try fewer orderings than a multiprocessor, they are not limited to one ordering, nor do they provide weaker guarantees regarding race conditions. Although simulators may try a smaller percentage of the orderings, the percentage executed by a multiprocessor is often so small as to provide no evidence against race conditions.

3.4 Simulators Require Small Data Sets

The performance of simulators typically promotes small data sets. The amount of memory on the workstation is also a limiting factor. However, this is not a limitation in practice, since most code is developed this way currently, i.e., using small partitions of a larger multiprocessor.

Although these data sets are small compared to production runs, they are usually large enough to test the code, probably even to tune it. For example, 10,000 operations on a B-tree is small compared to production expectations, but plenty for measuring the steady-state performance. Likewise, running successive over-relaxation (SOR) on a 1024x1024 grid on an 8x8 mesh places a 128x128 block on each processor, which is large for development purposes. Both simulations took about 2 minutes.[2]

[2] There were 20 iterations for the SOR PDE solver.

Although these two applications could be tuned on the simulator, the Water application from the SPLASH benchmarks probably could not. This program probably requires a large number of particles for successful tuning, and thus is beyond the scope of simulation (on a uniprocessor) *for performance tuning*. However, the smaller particle counts are quite sufficient for debugging.

A mitigating factor on the effects of small data sets is that it is typically much harder to get small data sets to perform well. For most applications, the relative cost of communication decreases as the problem size increases, which makes efficient communication less important. Thus, programs that run well on small data sets often run as well or better with larger data sets.

3.5 No Source-Code Compatibility

There are no fundamental problems here; PROTEUS and Tango already run parallel C with libraries and support other languages. (Tango also supports Fortran.) The only problem is ensuring that the libraries provide the same semantics and costs as those of the machine. This is time consuming but not difficult. Section 5.3 covers these issues in more detail.

3.6 Summary of Disadvantages

Except for source-code compatibility, none of the disadvantages are problematic for the development of most applications. There will be applications that need huge amounts of memory during development, applications that must run for billions of cycles even for development runs, and applications that have unusual vector unit or OS behavior, but we believe that these applications are rare. For such applications, perhaps only early development should be done via simulation.

For most applications, the disadvantages of simulation are inconsequential for most of the development process; the number and variety of applications that have *already* been developed on simulators supports this claim. We have shown that some applications can even be tuned on simulators, although we expect that many applications must be tuned on the multiprocessor.

Finally, technology trends are "on our side." As workstation performance increases, the performance of simulation and thus the achievable size of data sets will increase, which will improve both the quality and range of development via simulation.

4 Proposed Methodology

Although there are advantages to development via simulation, we are not advocating that all development be done on simulators, only that most development and perhaps some tuning exploit simulation. Thus, switching back and forth between the simulator and the real machine should require only recompilation. We view the correct methodology to be:

1. Perform initial development on the simulator, which includes selecting and implementing data structures and algorithms, and modular testing.

2. Continue with the simulator for most debugging and testing, and perform some profiling and tuning. This includes changing algorithms to remove bottlenecks and optimizing the critical procedures.

3. Once confident that the application is working well, switch to the real machine for more testing and final tuning. If tuning changes are likely to introduce bugs, then perform some regression testing on the simulator.

4. Start production runs, possibly with logging for replay. If a bug appears, use either the multiprocessor's debugger or switch to the simulator and use the log to guide the simulation. It may be advantageous to switch back and forth several times.

In the first part of this paper, we have argued that development via simulation is viable and often preferable. Given that some development and all production runs occur on the multiprocessor, the key goal is smooth integration of the simulator into the multiprocessor's development environment. The next section deals with issues of effective integration.

5 Making It All Work

There are several software engineering problems that should be addressed to make development via simulation more attractive. Three important issues are integrated support for replay from a multiprocessor log, extensions to sequential debugging for simulated parallel programs, and common libraries and compilers to allow easy movement of applications between simulators and the target multiprocessor.

5.1 Logging and Replay

Although logging and replay can provide repeatability on multiprocessors, their use involves several problems. First, the overhead of the logging may be very significant. Even with optimal logging, the overhead is typically 8–14% for message-passing programs and significantly more for shared-memory programs [NM92]; it can be as high as 600% for systems that trace some data for every message. A related problem with logging is the potentially huge trace files that are produced. Finally, the logging code is intrusive and thus alters the behavior of the program [Gai86], although hardware mechanisms exist that can nonintrusively produce some kinds of traces [LLG+92].

Despite these problems, logging and replay form the standard technique for achieving repeatability on multiprocessors. Thus it is important to be able to guide the simulator from a multiprocessor log. Fortunately, this is not difficult — it is nearly identical to replaying the program on the multiprocessor. Therefore, the multiprocessor should provide a logging and replay mechanism that is integrated with the simulator. The application will need recompilation to run on the simulator, but the log still exactly defines the order. Thus, if a logged program reveals incorrect behavior, the simulator can repeat that behavior until the bug is found.[3] A user with the log of an incorrect program can either debug the program on the multiprocessor or on the simulator, depending on the running time and memory requirements of the program (and the cost effectiveness of the platforms).

5.2 Extending Sequential Debuggers

Sequential debuggers are designed for only one processor and only one thread of control. Thus, PROTEUS extends sequential debuggers to handle parallel aspects of the application. The additional debugging support complements sequential debuggers such as **gdb**; users regularly switch between the two. PROTEUS' additional debugging tools fall into five categories:

Multithreading Although PROTEUS runs as a single process, it uses lightweight threads to achieve parallelism. Unlike **gdb**, which can trace only the current thread, PROTEUS can generate stack traces (with arguments, source files, and line numbers) for any thread. Furthermore, the use of many small threads increases the likelihood of stack overflow, so PROTEUS performs nonintrusive stack checks for early detection of stack overflow.

Access to Parallel Aspects of the Target In addition to threads, users need intelligent access to processors, messages, and shared memory. PROTEUS' internal debugger provides this access: users can examine the state of each processor including the ready queue, examine messages and pending interrupts, and access shared memory.

Synchronization Synchronization problems are a frequent source of bugs and need special support. PROTEUS allows users to examine the state of synchronization objects, including which threads are trying to acquire a lock (whether blocked or spinning) and the amount of contention. In cases of deadlock, this information can be used to track down the dependency cycle. PROTEUS also detects livelock for threads spinning on locks. All of this support is nonintrusive and probably not feasible on a real machine.

Overwrite Detection Overwrite bugs occur when a thread overwrites something it should not be modifying; out-of-bounds array modifications are an example. These bugs are difficult to track down in sequential applications and multithreading exacerbates the problem. The standard approach for these bugs is to trace access to the overwritten address, which is typically done by trapping after every statement to see if it modified the location. Although this works, the trap overhead reduces performance by a factor of 200 or more (in Unix). This is painful for any application and unbearable for long running simulations.

PROTEUS provides a two-step solution. First, the offending modification is narrowed down to the enclosing simulation event. This entails a quick check of the overwritten location per several hundred instructions and thus has little effect on performance. The simulation is then rerun up to that simulation event, at which point we use normal tracing (provided by the sequential debugger) to locate the overwrite bug. By using two steps, we incur the factor of 200 only for a few hundred instructions. Assuming the bug is in the middle of the simulation on average, this reduces the slowdown factor from about 100 (half the program at 200 times slower) to about $1 + \epsilon$ (two runs of half the program at nearly full speed).[4]

Graphics Although graphical debugging tools are useful for sequential programs, they are almost mandatory for highly parallel applications. They communicate information to the user with much higher bandwidth than text, and can help the user track the state of many processors and threads. PROTEUS provides integrated data collection and display facilities that regularly aid debugging. These tools are neither novel nor limited to workstations; nonetheless, we found graphical output to be an important extension of the sequential debugger for parallel applications. Figures 1 and 2 show graphs produced by PROTEUS.

In general, multiprocessors and simulators should share data collection and display tools. Such tools typically interpret trace files generated by the application; the simulator could easily generate the same trace format as the multiprocessor. In fact, the graphics tools provided with PROTEUS have been used with logs generated by real multiprocessors.

5.3 Source-Code Compatibility

Ideally, the compiler for the real machine should also output code (low-level C or Fortran) for the simulator.

[3] As expected, an unlogged program may or may not repeat the incorrect behavior when rerun, rerun with logging, or rerun on the simulator.

[4] A two-step method for handling overwrite bugs should be provided for all (repeatable) applications, not just simulators. Of course, some languages prevent these kinds of bugs.

This would ensure that all of the high-level transformations and optimizations are the same, including data placement and alignment decisions. Thus, the vendor or a third party would provide the libraries for the simulator and the integrated compiler.

Currently, the primary languages supported by high-performance simulators are parallel versions of C and Fortran. This is consistent with nearly all multiprocessors. In addition, high-performance simulators already support key libraries and macro packages, such as the ANL macros [JBD+87], the Portable Instrumented Communication Library (PICL) [GHPW90], and active messages [vCGS92]. Programs written with these packages can already be moved between simulator and multiprocessor quite easily.

To provide source-code compatibility between PROTEUS and the CM-5, we are working with Thinking Machines to move the CMMD 3.0 [TMC93] interface to PROTEUS. This would allow most MIMD C programs to be fully source-code compatible. This compatibility, when combined with a CM-5 version of PROTEUS, should provide an excellent development environment.

The EPPP project [VSB+92] is already taking the next logical step: integrating support for simulation into the compiler. Their compiler can generate code either for the target multiprocessor or for PROTEUS. This level of integration provides tremendous accuracy because the PROTEUS code undergoes the same optimization, register allocation, and instruction scheduling phases as the code for the multiprocessor.[5] Switching between simulator and multiprocessor is essentially a command-line argument.

To summarize, the environment should at least be able to run the application on either the simulator or the multiprocessor. This means providing the same macro packages and libraries for both platforms, ideally with the same costs. Integrating support for simulation into the compiler takes this a step farther by generating nearly identical code for both platforms. In general, as the level of integration improves, the amount of development that *must* be done on the multiprocessor decreases.

6 Parallel Simulation

Applications that require huge amounts of memory, long running times, or very large data sets may not be feasible on a workstation. However, the repeatability, the support for nonintrusive monitoring, and the versatility of simulation can still be achieved for these applications if run on a parallel simulator. Parallel simulation forfeits the workstation advantages of simulation for the larger memory and greater speed of a multiprocessor. But unlike running on the multiprocessor directly, parallel simulation provides repeatability, nonintrusive monitoring and debugging, and the versatility to simulate many architectures. It thus has some place between simulation on a workstation and development directly on the multiprocessor.

Parallel simulation should achieve better accuracy than simulation on a workstation for vector units and memory-mapped I/O operations, since these operations cannot be done directly on the workstation. If the simulator is simulating the host multiprocessor, then integrated compiler support for simulation comes for free.

A variation of parallel simulation called *virtual prototyping* is already in use in the Wisconsin Wind Tunnel (WWT) [RHL+93]. This system runs on a CM-5 and uses direct execution to simulate everything except cache misses. Cache misses are simulated by causing an error-correcting code trap on accesses to shared cache lines. The trap handler simulates the cache coherence protocol in detail. Currently, other forms of interaction such as message passing are not supported, although there are no fundamental obstacles.

The parallel version of PROTEUS, which is currently under development, is similar to the Wisconsin Wind Tunnel in that both use conservative parallel discrete-event simulation techniques [Fuj90] to keep the processors synchronized to within a specified number of simulated cycles. Parallel PROTEUS, however, is more decoupled from the host and can simulate a wider variety of architectures, including accurate simulations of many interconnection networks.

Decoupling Parallel PROTEUS from the host could produce substantially worse performance than virtual prototyping, except that the dominant cost for both systems is keeping the nodes synchronized. In fact, for problems that fit in the memory of one workstation, the existing sequential version of PROTEUS appears to be competitive with or faster than the published numbers for WWT, even though WWT runs on a CM-5. This is due to the synchronization overhead required by conservative parallel discrete-event simulation. This overhead is amortized as the number of simulated processors increases, so the performance of WWT improves as the simulated machine increases in size. Performance models for Parallel PROTEUS predict similar behavior: the parallel version will be useful primarily for large data sets, large memory requirements, or large numbers of processors (more than a few hundred).

Thus, parallel simulators combine the power and memory of multiprocessors with most of the advantages of simulation. They allow simulation of very large data sets or very large machines — performance models indicate that Parallel PROTEUS could reasonably simulate machines with 100,000 nodes on a 512-node CM-5. However, for small data sets and less than 100 nodes, the workstation simulator is usually faster, due to the large startup cost of synchronization for (conservative) parallel simulation. In the long run, all three options — workstation simulation, parallel simulation, and using the multipro-

[5] They use workstations with the same processor as the multiprocessor nodes.

Development Issues	Kind of Simulator Sequential	Parallel	Multiprocessor
Workstations are cost effective	Yes	No	No
Multiprocessors are a scarce resource	Yes	No	No
Workstations are always available	Yes	No	No
Repeatability	Yes	Yes	With Logging
Use of sequential debuggers	Yes	Yes[a]	No
Users have access to all state	Yes	Yes[b]	No
Nonintrusive debugging/monitoring	Yes	Yes	No
Measure all aspects of target	Yes	Yes	No
Versatility — simulate many targets	Yes	Yes	No

a: Simple versions of `dbx` already run on some multiprocessors, but much better versions could be run on top of the parallel simulator. It should be relatively easy to build a Parallel Proteus back end for `gdb`.

b: Although the application may run in several address spaces, a coherent view of all state can still be obtained since the simulation can be stopped at a particular point. This kind of access could be built into the `gdb` back end.

Table 2: Summary of simulation advantages over running directly on a multiprocessor.

cessor directly — should be integrated. An application should be runnable on all three, and different scenarios will prefer different options.

Finally, we actually practice what we preach: the parallel version of PROTEUS is being developed on top of PROTEUS. Currently, we can run simple programs on the parallel simulator, which itself is an application running on PROTEUS. Later, we will move the parallel simulator to the CM-5. Key data structures, like the concurrent priority queue, have already been tested and tuned; we expect little trouble when we use them on the CM-5.

7 Summary

Application development via high-performance simulation offers many advantages. Simulators can provide a flexible, cost-effective, interactive debugging environment that combines traditional debugging features, such as repeatability and powerful debugging tools, with completely nonintrusive data collection for debugging or profiling. Table 2 summarizes the advantages provided by simulators running on workstations and multiprocessors.

The only remaining obstacle to such an environment is source-code compatibility: the simulator should provide the same interface and costs as the target multiprocessor. Toward this goal, we are developing a version of PROTEUS for the CM-5 that will support key libraries and thus provide source-code compatibility for most C applications. Our future CM-5 development, which includes the parallel version of PROTEUS, will occur on PROTEUS — only production runs and tests with very large data sets will actually take place on the CM-5.

Acknowledgements: Thanks to Wilson Hsieh, Anthony Joseph, David Keppel, Ulana Legedza, Eric Reiher, Martin Serrano, Brad Spiers, Pearl Tsai, Carl Waldspurger, Paul Wang, and Donald Yeung.

[BD92] E. A. Brewer and C. N. Dellarocas. PROTEUS *User Documentation*. Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, December 1992. Version 0.5.

[BDCW91] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS*: A High-Performance Parallel-Architecture Simulator*. MIT Technical Report MIT/LCS/TR-516, September 1991.

[BDCW92] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. In *Proceedings of the 1992 ACM SIGMETRICS and PERFORMANCE '92 Conference*, pages 247–248, June 1992.

[Bed90] R. Bedichek. Some efficient architecture simulation techniques. In *Proceedings of Winter 1990 Usenix Conference*, January 1990.

[BL92] T. Ball and J. R. Larus. Optimally profiling and tracing programs. In *Proceedings of the 1992 Principles of Programming Languages (POPL '92)*, pages 59–70, January 1992.

[CBDW91] A. Colbrook, E. A. Brewer, C. N. Dellarocas, and W. E. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, pages III138–III141, August 1991.

[CDJ+91] R. G. Covington, S. Dwarkadas, J. R. Jump, J. B. Sinclair, and S. Madala. Efficient simulation of parallel computer systems. *International Journal in Computer Simulation*, 1:31–58, 1991.

[Cha90] D. Chaiken. Cache coherence protocols for large-scale multiprocessors. Technical Report MIT/LCS/TR-489, MIT Laboratory for Computer Science, September 1990.

[CJS89] R. G. Covington, J. R. Jump, and J. B. Sinclair. Cross-profiling as an efficient technique in simulating parallel computer systems. In *Proceedings of the 1989 Conference on Computer Software and Applications (COMPSAC '89)*, pages 75–80, 1989.

[CLN90] D. Chaiken, B.-H. Lim, and D. Nussbaum. ASIM Users Manual. ALEWIFE SYSTEMS MEMO #13, MIT Lab for Computer Science, August 1990.

[CMM+88] R. G. 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*, May 1988.

[Col] A. Colbrook. Personal Communication.

[DBMI88] G. A. Darcy, III, R. F. Brender, S. J. Morris, and M. V. Iles. Using simulation to develop and port software. *Digital Technical Journal*, 4(4), 1988. Special Issue on Alpha AXP.

[DGH91] H. Davis, S. R. Goldschmidt, and J. Hennessy. Multiprocessor simulation and tracing using Tango. In *Proceedings of the 1991 International Conference on Parallel Processing (ICPP '91)*, pages II99–II107, August 1991.

[FC88] R. Fujimoto and W. B. Campbell. Efficient instruction-level simulation of computers. *Transactions of the Society for Computer Simulation*, 5(2):109–124, 1988.

[Fuj90] R. M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[Gai86] J. Gait. A probe effect in concurrent programs. *Software – Practice and Experience*, 16(3):225–233, March 1986.

[GHPW90] Geist, Heath, Peyton, and Worley. A user's guide to PICL: A portable instrumented communication library. Technical Report ONL/TM-11616, Oak Ridge National Laboratory, 1990.

[HM92] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. Technical Report CRL 92/07, Digital Equipment Corporation, December 1992.

[HWW93] W. Hsieh, P. Wang, and W. E. Weihl. Computation migration: Enhanced locality for distributed-memory systems. In *Proceedings of the 1993 PPoPP*, May 1993.

[JBD+87] Boyle J., R. Butler, T. Disz, B. Glickfeld, E. Lusk, R. Overbeek, J. Patterson, and R. Stevens. *Portable Programs for Parallel Processors*. Holt, Rinehart, and Winston, New York, 1987.

[Lar93] J. Larus. Basic-block profiling isn't always accurate. Usenet newsgroup `comp.compilers` article, March 24, 1993.

[Lin90] M. A. Linton. The evolution of dbx. In *Proceedings of the 1990 USENIX Summer Conference*, pages 211–220, June 1990.

[LLG+92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessey, M. Horowitz, and M. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.

[LMC86] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with Instant Replay. Technical Report TR194, University of Rochester, Computer Science Department, September 1986.

[MCS91] J. M. Mellor-Crummey and M. L. Scott. Synchronization without contention. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 269–278, April 1991.

[MF88] I. Mathieson and R. Francis. A dynamic-trace-driven simulator for evaluating parallelism. In *Proceedings of 21st Hawaii International Conference on System Sciences*, volume 1 (Architecture), pages 158–166, January 1988.

[NDD93] G. Nelson, D. Detlefs, and D. Doligez. DEC Systems Research Center, Work in progress, 1993.

[NM92] R. H. B. Netzer and B. P. Miller. Optimal tracing and replay for debugging message-passing parallel programs. In *Proceedings of Supercomputing '92*, pages 502–511, August 1992.

[Qui89] M. J. Quinn. Analysis and benchmarking of two parallel sorting algorithms: Hyperquicksort and quickmerge. *BIT*, 29(2):239–250, 1989.

[RHL+93] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, May 1993.

[SP92] R. M. Stallman and R. H. Pesch. *Using GDB: The GNU Source-Level Debugger*. Free Software Foundation, 4.04 edition, 1992.

[SPA92] SPARC International. *The SPARC Architecture Manual, Version 8*, 1992. Prentice Hall, Englewood Cliffs, NJ.

[SWG92] J. P. Singh, W.-D. Weber, and A. Gupta. SPLASH: Stanford parallel applications for shared memory. *SIGARCH Computer Architecture News*, 20(1):5–44, March 1992.

[TLK92] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, March 1992.

[TMC92] Thinking Machines Corporation. *The Connection Machine CM-5 Technical Summary*, January 1992.

[TMC93] Thinking Machines Corporation. *CMMD 3.0*, February 1993.

[vCGS92] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrated communication and computation. In *Proceedings of the 19^{th} International Symposium on Computer Architecture (ISCA '92)*, pages 256–266, May 1992.

[VSB+92] V. Van Dongen, A. Singh, C. Bonnello, M. Hancu, G. Gao, L. Hendren, P. Panangaden, H. Hum, L. Tao, and M. Feeley. Towards a portable parallel programming environment. In *Proceedings of the 1992 Supercomputing Symposium*, Montreal, June 1992.

[WBC+91] W. E. Weihl, Eric Brewer, Adrian Colbrook, Chrysanthos Dellarocas, Wilson Hsieh, Anthony Joseph, Carl Waldspurger, and Paul Wang. PRELUDE: A system for portable parallel software. Technical Report MIT/LCS/TR-519, Massachusetts Institute of Technology, October 1991.

[WW93] C. A. Waldspurger and W. E. Weihl. Register relocation: Flexible support for multiple contexts. In *Proceedings of the 20^{th} International Symposium on Computer Architecture (ISCA '93)*, May 1993.

[Yel92] K. Yelick, September 1992. Personal Communication.