A Worksation-Based Parallel Direct-Execution Simulator*

Phillip M. Dickens
Department of Computer Science and Applied Mathematics
Illinois Institute of Technology

Abstract

It is important to understand and efficiently predict the behavior of large message-passing parallel programs executing on massively parallel machines, especially as these machines are becoming a dominant force in the scientific computing arena. One important tool for this kind of research is direct-execution simulation, which provides performance and scalability analysis of application codes executing on top of some *virtual computer* of interest. We have developed such a tool called LAPSE (Large Application Parallel Simulation Environment), which is implemented on the Intel Paragon and provides excellent performance predictions for a wide variety of codes executing on a virtual Paragon. However, massively parallel machines such as the Paragon are scarce, expensive, and generally unavailable to large segments of the research community, making it important to implement such tools on top of computational platforms more widely available to the research community at large. To meet this need, we have ported LAPSE to a workstation cluster using the **nx_lib** message passing library, which provides the *functionality* of a Paragon on a network of workstations. The goal of this research is to provide researchers the opportunity to study the behavior of large parallel codes, written for massively parallel machines such as the Intel Paragon, while physically executing on a workstation cluster.

This research is being carried out in two phases. The first phase, which is now complete, involves improving the performance of LAPSE such that it is feasible to execute large simulations on a workstation cluster. In this paper, we describe our technique for reducing execution costs, and demonstrate up to a four fold increase in performance. The second phase, which is now in its infancy, involves investigating mechanisms to augment LAPSE on a workstation cluster with *temporal* capabilities: the ability to predict execution times for a code had it been executed on the Paragon. While we have no results to present as of yet, we do describe the approach we intend to pursue to solve this difficult problem.

1 Introduction

Direct-execution simulation is an important tool for performance and scalability analysis of application codes executing on top of some presumed *virtual computer* of interest. In this approach, a *host* computer is used to simulate the behavior of an application code executing on top of a virtual, or *target* machine. If the host and target machine are of the same architecture, this similarity can be exploited by simulating only those details of the virtual machine not available on the host machine. For example, a new operating system, a new cache coherence protocol, or a new communication network would require

*This research was also supported by the National Aeronautics and Space Administration under NASA contract NAS1-19480 while the authors were in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23681-0001.

simulation; the execution of the native instruction set would not. Thus the application code executes natively *until* it has an interaction with the simulated virtual machine, at which point the application traps out, and a discrete-event simulator models the impact of the particular call on the state of the virtual machine. From the point of view of the application, it *is* executing on the virtual machine. From the point of view of the discrete-event simulator, the application is a driver, describing activity to be simulated.

A detailed direct-execution simulator is computationally intensive, particularly when simulating a machine with multiple processors. However, these costs can be reduced by parallelizing the application code *and* the discrete-event simulator. We have developed such a parallel-direct execution simulator called LAPSE (Large Application Parallel Simulation Environment), where both the application code and the virtual machine simulator are parallelized.

LAPSE is implemented on the Intel Paragon, and provides excellent performance predictions for a wide range of application codes executing on top of a virtual Intel Paragon [4]. However, massively-parallel machines such as the Intel Paragon are scarce, expensive, and multiplexed among many users, making it difficult (if not impossible) for researchers to gain access to a large number of processors on a regular basis. It is therefore important to implement performance analysis tools for such massively parallel machines on top of computational platforms more accessible to the research community at large. To meet this need, we have ported LAPSE to a workstation cluster using the **nx_lib** message passing library [13], which provides the *functionality* of a Paragon on a network of workstations. The goal of this research is to provide researchers the opportunity to study the behavior of large parallel codes, written for massively parallel machines such as the Intel Paragon, while physically executing on a workstation cluster.

This research is being carried out in two phases. The first phase addresses the very poor performance of LAPSE on a workstation cluster. The problem stems from the implementation of the virtual processors of the simulated system. LAPSE models the Intel Paragon, a distributed memory machine, where each processor has its own private address space. Given this, the virtual processors of the simulated system must also have (at least) the functionality of a private address space. The most natural way to accomplish this is to implement the virtual processors using heavy-weight Unix processes. However, this approach results in very high on-processor communication costs and context switching costs.

The performance of LAPSE on the Intel Paragon is quite reasonable even though each processor may perhaps multiplex a large number of Unix processes. This is because the Paragon enjoys many excellent performance features, including an extremely fast interconnection network which minimizes off-processor communication costs. However, when moving to a workstation cluster connected by ethernet, both on-processor *and* off-processor communication costs are quite high, severely impacting the performance of LAPSE on this platform. In fact, the performance is so poor that it makes the execution of large simulations impractical. We sought to address this issue before attempting to obtain performance estimates for large codes intended for execution on the Intel Paragon.

The first phase of this research is now complete, and we present the results of this research here. To improve performance, we developed a simple technique to implement the virtual processors of the simulated system using light-weight threads rather than heavy-weight Unix processes, reducing both on-processor communication costs and context switching costs. What makes this a difficult problem is that each virtual processor requires the *functionality* of a separate address space, and threads are generally implemented such that they all share a global address space. Other researchers have investigated this issue. The advantage of our approach is that it provides each thread with the *functionality* of a private address space without special operating system support (as required by the Wisconsin Wind Tunnel project [12]), without access to the base address register (as required in the User Level PVM [7]), and with minimal modifications to the application code.

Details of our approach can be found in [6], and we only provide a brief review below. In this paper, we are interested in quantifying the speedup that can be obtained using this simple technique. We use an iterative, synchronous, scientific code to compare the performance of the thread-based and process-based implementations on LAPSE. We investigate relative speedup as a function of the number of processors on which the simulation executes and the computation to communication ratio.

The second phase of this research is not yet complete, and we therefore have no results to report. It is a difficult issue to derive performance estimates for a multicomputer system with one CPU, using a direct-execution simulator executing on another multicomputer system with a different type of CPU. In fact, we know of no other project attempting to do so. However, we have identified a very simple extension to a mechanism currently employed by LAPSE which may provide the functionality we seek. We briefly outline this proposed approach at the end of the paper.

Before leaving this section, it is important to note that several other projects use direct execution of application processes to drive simulations of multiprocessor systems [1, 2, 3, 5, 11]. The Wisconsin Wind Tunnel (WWT) [11] is, to our knowledge, the only multiprocessor simulator that uses a multiprocessor (the CM-5) to execute the simulation. It is worthwhile to note the differences between LAPSE and WWT. The first is the issue of purpose of the system. The WWT is a tool for cache-coherency protocol researchers. LAPSE is designed primarily for performance and scalability analysis. LAPSE is implemented on an Intel Paragon and a cluster of Sun Sparc workstations, neither of which supports shared virtual memory. Coherency protocols complicate the the simulation problem considerably, but are a facet we are not dealing with currently. A second difference is the synchronization mechanism used to maintain the fidelity of the parallel simulation. WWT uses a special case of the YAWNS synchronization protocol [9] while LAPSE uses a synchronization mechanism combining YAWNS with appointments [8]. We do not elaborate on the details of the synchronization mechanism here. The interested reader is directed to [4] for a detailed discussion. The third difference is in the implementation of virtual processors using light-weight threads. In particular, the approach taken in [11] requires significant operating system support while our approach does not. We discuss this issue in more detail below.

The remainder of the paper is organized as follows. In Section 2 we briefly review the implementation of LAPSE on the Intel Paragon and on a cluster of workstations. In Section 3, we describe our experimental method and results. In Section 4, we outline a very simple approach we intend to pursue to obtain the performance predictions we seek, and we give our conclusions in Section 5.

2 LAPSE Overview

LAPSE models the behavior of a virtual Intel Paragon. The Paragon is a distributed memory machine, where the processors communicate exclusively via message passing over a mesh connected communication network. In such a network, each processor has a direct communication link between itself and its neighbors to the north, south, east and west. Figure 1 depicts an Intel Paragon with sixteen processors.

At the software level, the processors communicate and synchronize exclusively through the **nx_lib** message passing library. This library provides a rich set of communication primitives including functions to send messages, receive messages, probe to determine if a particular message is available, wait for a

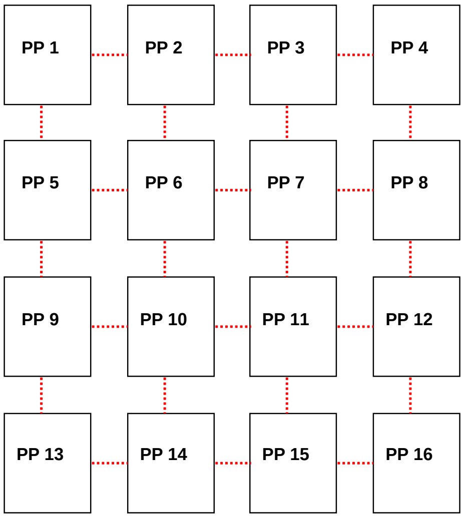

Figure 1: A Sixteen Node Intel Paragon

message, or enter into a global reduction operation. It is important to note that *all* message passing activity is explicitly initiated by calls to this library.

At any given time, an application process will be engaged in either performing some computation over its private data set (termed an execution burst), or executing some call to the communication library (called a communication burst). Thus we can view the progress of an application process as an execution burst of some unknown duration followed by a communication burst of some unknown duration. This is depicted in Figure 2.

Given N application processes whose performance on N processors is sought, we use $n <= N$ processors to both execute the application code and simulate its timing behavior. Each processor is assigned some number of application processes (also termed *virtual processors* or *VPs*), and one simulator process. The simulator processes control the execution of the application processes, and, together with other simulator processes, emulate the behavior of the assumed virtual machine. The execution behavior of the application processes executing on the virtual machine is obtained by actually running the application processes—all interactions of the application processes with the virtual machine are trapped, and the impact of that interaction on the state of the virtual machine is modeled by the simulator processes.

To help capture temporal behavior, LAPSE gets execution time information from the application processes. At compile-time, LAPSE modifies the application assembly code, inserting instruction counters at basic block boundaries. Whenever an application process makes a call to **nx_lib**, it represents an interaction with the virtual machine, and thus the call traps out to the simulator process managing its execution. The simulator process recovers the number of instructions executed since the last such interaction with the virtual machine, which represents an estimate of the amount of time the process spent

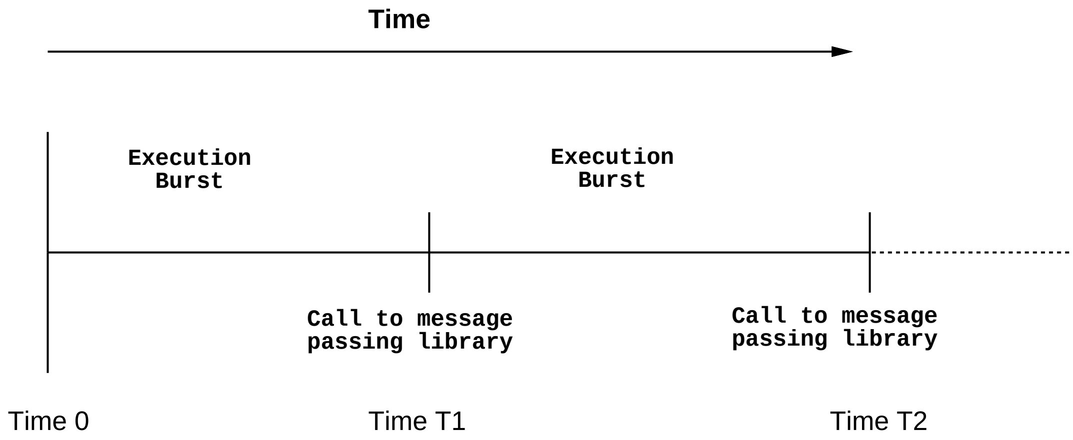

Figure 2. Execution Pattern of an Application Process

Figure 2: Execution Model of an Application Process

in the current execution burst. The simulator processes cooperatively determine how time advances in the simulated system as a function of the estimates of the duration of execution bursts, assumed operating system overheads for message-passing and interrupt handling, a model of the communication network of the assumed virtual machine, and a model of how message-passing activity impacts a virtual processor.

It is possible for the simulation activity on one processor to affect the activity on another, and we thus need a synchronization mechanism to control the execution of the simulation. LAPSE uses a conservative protocol called *Whoa* (window based halting on appointments), which combines the YAWNS protocol with appointments. A detailed discussion of our approach, and the synchronization protocol used to guarantee the fidelity of the simulation, can be found in [4].

2.1 LAPSE on a Workstation Cluster

Our port of LAPSE to a workstation cluster uses the **nx_lib** software library [13], which allows the development and execution of codes written for the Intel Paragon using an ordinary network of workstations. As noted, our initial implementation of LAPSE allocated one heavy-weight Unix process for each virtual processor. Additionally, LAPSE implements each simulator process with a heavy-weight Unix process.

Under **nx_lib**, all inter-process communication is performed through Unix sockets. LAPSE can be communication intensive since the application processes interact with each other as well as with the simulator processes. Additionally, the simulator processes frequently communicate to share information and to synchronize. Since each such communication goes through a Unix socket, the cost of both on-processor and off-processor communication is quite high, severely degrading the performance of LAPSE on a workstation cluster.

```
#include <stdio.h>

double foo1() ;

int func1() ;

int x, y, z ;

main()
 {int a, b, c ;

double d1 ;
a = x ; b = y ;
z = a + b ;
x = z ;
d1 = foo1() ;
......
}

double foo1()
{ ..... }

int func1()
{.....}
```

Virtual Processor Code

```
#include <stdio.h>
  class G_Class {
    public:
      int x, y, z ;
   double foo1() ;
     int func1() ;
     new_main() ;
     } ;

G_Class::new_main()

 { int a,b,c ;
   double d1 ;
   a = x ; b = y ;
   z = a + b ;
   x = z ;
   d1 = foo1() ;
   .......
 }

Thread_Entry_Function()

{ G_Class  my_class ;

  my_class.new_main() ;
}
```

Separate Addess Spaces

Figure 3: Creating Separate Address Spaces for Light-weight Threads

To reduce these costs, we have modified LAPSE to implement the application processes with light-weight threads rather than heavyweight Unix processes. This reduces on-processor communication to a memory copy, and also reduces the costs associated with context switching.

Implementing application processes as lightweight threads requires some mechanism to provide each thread with the functionality of its own private address space. However, this presents a problem since threads are generally implemented such that they all share a global address space. We solved this problem by employing the C++ mechanism for class scope to provide thread-specific address spaces that are independent of both the underlying thread system and the target machine.

Recall that in C++, class definitions have their own scope, which means that variables (data members) defined within a class are visible only through the class pointer, and are maintained separately for each instance of the class. Thus, to provide thread specific data in our system, the global data and functions, including the main function, are surrounded by a class definition, creating a global class. Each thread allocates a single instance of this object. Since all the functions are transformed into member functions of this class, all references to global data are automatically transformed by the C++ compiler to indirectly reference the class pointer, `this`. To make these ideas more concrete, consider the example shown in Figure 3.

The column on the left side of the figure shows the original application code that is to be executed by each virtual processor. There are three global variables (x,y,z) and two global functions (foo1 and func1) all of which should be global within each thread.

Now consider the right column of the figure which shows the modifications to the user code necessary to give each thread its own address space. First, all of the global variables, global functions, and the function **main()** (renamed **new_main()**) of the application code are put into one class (**G_Class** in this example). Second, a thread entry function is written which declares an object of type **G_Class** and calls the renamed main function for this object (**my_class** in this example). Note that when the **new_main()** function is called, it is executing as a member within the class **G_Class**. Thus, when it accesses a previously global variable it is actually accessing its own private version of the variable through the implicit **this** pointer. A new main function has to be written (not shown here) which creates the N threads, one for each virtual processor, using the thread entry function shown above. At this point each thread has its own "private" address space via the class **G_Class**.

Each thread allocates a single instance of this object. Since all the functions are transformed into member functions of this class, all references to global data are automatically transformed by the C++ compiler to indirectly reference the class pointer, `this`.

The interested reader is directed to [6] for a detailed discussion of our approach and the approaches taken by other researchers.

3 Experimental Design

We are interested in quantifying the speedup that can be obtained using our simple approach. To get a measure of this, we employ an iterative, synchronous, scientific code we will call **SOR**. **SOR** solves Poisson's boundary equations in two dimensions (see Chapter 17 in [10]) and uses a NEWS (North East South West) communication pattern. All message passing is synchronous. Like many numerical problems, **SOR** uses a large two dimensional array and is iterative in nature. Each iteration is composed of three steps: A communication phase, a computation phase and a synchronization phase. The opportunity for parallelism lies within the computation phase where all of the processors operate on different pieces of the matrix in parallel.

A barrier synchronization is required between each iteration because values generated in the previous iteration are used in the current one. The program converges to a solution whose accuracy is a user specified parameter. At the end of each iteration, the processors collectively determine the maximum error term taken over all processors in the system. When this error term is less than the input parameter the program is completed.

A key parameter of **SOR** is G, where each processor is responsible for a GxG subgrid of the matrix. Assuming the number of processors stays constant, increasing G will increase the computation to communication ratio, and decreasing G reduces this ratio. In the results presented here, we consider a grid size of $G = 25$ to be a low computation to communication ratio, a grid size of $G = 100$ to be a medium computation to communication ratio, and a grid size of $G = 250$ to be a high computation to communication ratio.

We chose this application code for two reasons. First, it is representative of a large class of scientific numerical codes. Secondly, it gives us the ability to study the effects on performance of varying the computation to communication ratio.

The system we simulate has a constant 32 virtual processors, and we use 1,2,4,8 and 16 physical processors to execute the simulation. All experiments were executed on a network of Sun Sparc 20s. The normalized metric of performance is the number of seconds required to complete one iteration of the algorithm. Each data point represents the mean of ten trials.

3.1 Experimental Results

There are two expected benefits of using light-weight threads: reduced communication costs for threads executing on the same processor and reduced context switching costs. Light-weight threads will of course do nothing to reduce computation costs. Thus we would expect threads to have the greatest impact on performance when the computational load is low, and there are many virtual processors executing on the same physical processor. Also, we expect that as the computation to communication ratio increases, this relative advantage decreases.

In Figure 4, we plot the seconds per iteration of the algorithm for both the thread-based and process-based approaches as we vary the number of processors between one and sixteen. In this experiment, we use a small computation to communication ratio ($G = 25$). As expected, the improvement in performance is greatest when all 32 virtual processors are executing on one physical processor. In this case, the thread-based approach increases performance by a factor of 4.15. As can be seen, both approaches benefit from increasing the number of processors allocated to the simulation (except as noted below). However, as the number of processors increases, the relative advantage of the thread-based approach begins to decrease. With sixteen processors, the relative improvement in performance falls from 4.15 to 3.28.

It is interesting to note that neither approach realizes any improvement in performance as the number of processors is increased from eight to sixteen. This has to do with the inverse relationship between extra computational power (i.e. more processors) and synchronization costs. As we double the number of processors from eight to sixteen, the communication and synchronization costs increase significantly. However, the additional eight processors do not have enough computation to perform to overcome the increased synchronization costs.

It is perhaps counter-intuitive that we observe an improvement in performance as large as 3.28 when sixteen physical processors are allocated to the simulation. However, with sixteen processors there are still two Unix processes per processor in the thread-based approach (one for the two threads and one for the simulator process), and three Unix processes per processor in the process-based approach (one for each of the virtual processors and one for the simulator process). Thus the thread-based approach can still have a positive impact on communication and context switching costs.

In Figure 5, we again plot the performance of the two approaches as we increase the number of processors from one to sixteen. In this case, we employ a medium computation to communication ratio ($G = 100$). We observe a significant improvement in performance when all 32 virtual processors reside on the same physical processor (a factor of 3.3). While significant, this improvement is not as great as that observed for $G = 25$: with a larger grid size the computation costs begin to dominate the communication and context switching costs, thus making the use of threads less important.

We again observe that both approaches benefit from increasing the number of processors, even as we move from eight to sixteen processors. In this case, the increased computational costs are high enough to offset the increased synchronization costs. As expected, we observe a decrease in the magnitude of the relative improvement as the number of processors increases. With sixteen processors, this relative improvement falls from 3.3 to 2.46.

In Figure 6, we look at the same information with a heavy computation to communication ratio ($G = 250$). As expected, both approaches benefit as we increase the number of processors allocated to the simulation from one to sixteen. Note that the improvement in performance for $G = 250$ is significantly less than that for $G = 25$ (an improvement of 1.94 versus 4.15 when all 32 threads reside

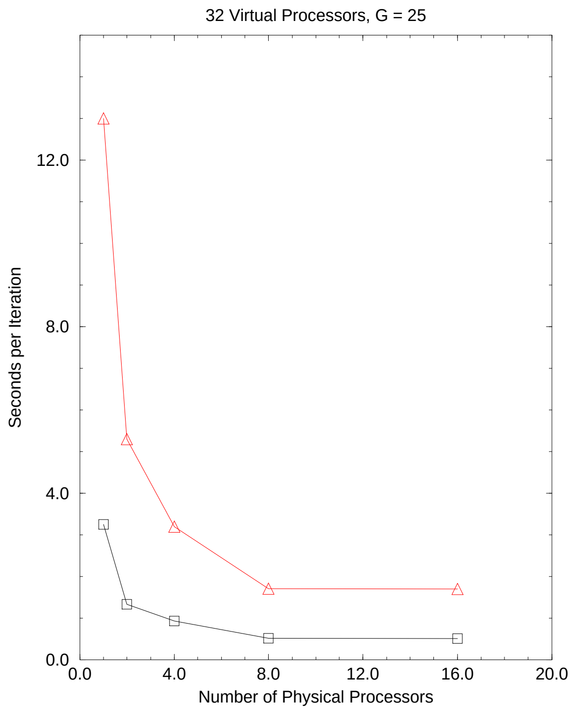

Figure 4: Performance measurements using 32 Virtual Processors with a Small Computation to Communication Ratio
.

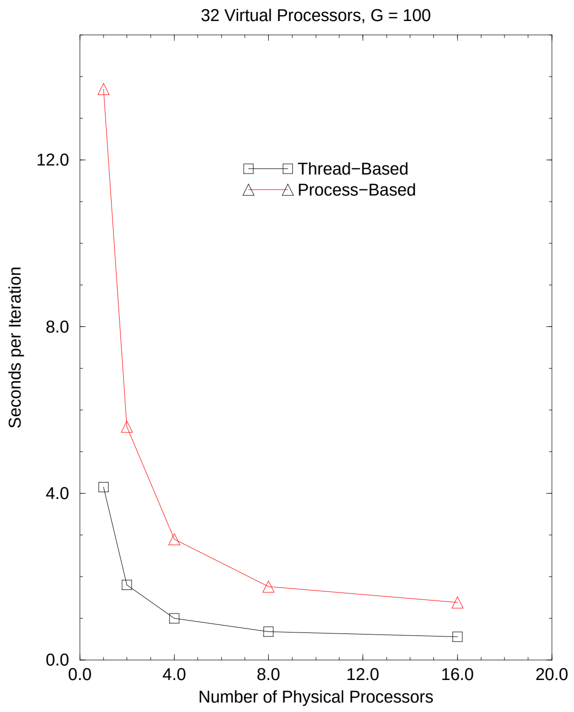

Figure 5: Performance measurements using 32 Virtual Processors with a Medium Computation to Communication Ratio

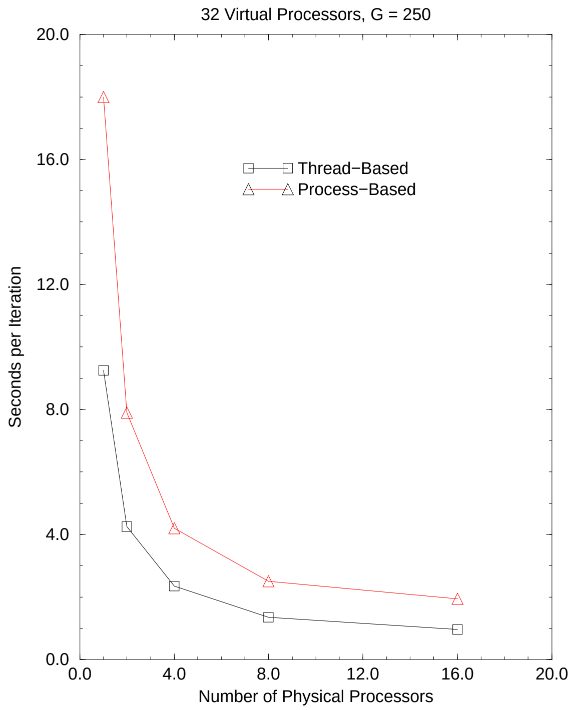

Figure 6: Performance measurements using 32 Virtual Processors with a High Computation to Communication Ratio

on the same processor). This reflects the fact that communication and context switching costs are dominated by the computational costs, reducing the impact of using light-weight threads.

4 Approach to Phase Two of the Project

As noted, **nx_lib** provides the *functionality* of a Paragon on a workstation cluster. However, it is possible for the execution path of a code to be different on a workstation cluster than it would be on the Intel Paragon. Furthermore, calls to system clocks reflect the workstation's own sense of time, not the time on the Paragon being simulated. Thus **nx_lib** *cannot*, in and of itself, provide accurate timing estimates of how long a code would have taken to complete had it been executed on the Paragon.

It is important to note that LAPSE on a workstation cluster already simulates *much* of the temporal behavior of the Paragon. That is, the communication network being simulated is the mesh connected network of the Paragon. The assumed operating system overheads are those that were measured on the Paragon. The model of the impact on the state of the virtual machine due to interrupt handling and other message-passing activity is that developed for the Intel Paragon. Thus almost all of the pieces necessary to model the temporal behavior of the Intel Paragon are already available. The temporal information *not* available is the mapping of the duration of an execution burst on a workstation cluster to the duration of that same execution burst on an Intel Paragon. The success of this project of course hinges on the assumption that such a mapping exists, and the bulk of the research will be geared towards establishing this mapping.

A very simple approach to capturing this last piece of temporal information may be to modify a mechanism already used by LAPSE on the Intel Paragon. Recall that LAPSE uses the number of instructions in an execution burst as an estimate of the time spent in that burst. We perform a very simple heuristic to map the number of instructions to the amount of wall clock time spent in the execution burst. First, we run the application natively (i.e. without LAPSE) on a small number of nodes (typically on the order of two to four), with a certain data size per node (e.g. G in the **SOR** application). We use the Unix **time** command to estimate U, the amount of user time consumed by the application code. We then run the application (with the same data size) under LAPSE, which reports I, the total number of user instructions. From this, we compute a *conversion factor* $c = U/I$, which is the average time per user instruction. This conversion factor incorporates, in an average case sense, the instruction mix, cache hit ratio, translation lookaside buffer hits and so forth of the application. The estimated cost per user instruction is then inflated by an amount determined by the conversion factor, and this inflated estimate is used in subsequent runs of the application under LAPSE for timing predictions.

This is an admittedly simple approach to calibration, and more accurate techniques could certainly be used. However, timing predictions based on this approach have been quite good for the codes we have tested [4]. Our initial approach to mapping the durations of the execution bursts of the two platforms will be to simply add another layer to the mechanism currently used. In particular, we will use our current technique to compute the average time per user instruction for a code executing on a workstation cluster. Then, we will execute the same code on the Intel Paragon (using a small number of processors), and determine the average time per user instruction for that platform. We will then compute a new conversion factor, C, the ratio of the two computed conversion factors. This ratio will then be used to modify the value of the average time per user instruction on the workstation cluster. For example, let A be the calculated average time per user instruction for the workstation cluster, and let B represent this value for the Intel Paragon. Then the new conversion factor is $C = A/B$. A would then be recalculated as $A = A/C$, thus adjusting for the difference in execution speed of the two platforms.

It may be the case that the hardware platforms, and the compilers for the two systems, are too dissimilar to allow for this kind of simple mapping. It may work quite well for some or many types of problems. This is an important area of current research.

5 Conclusions

This paper describes our investigation into developing performance prediction tools for a very scarce resource (an Intel Paragon), using a widely available resource (a network of workstations). The first issue we had to address was the very poor performance of LAPSE on a workstation cluster. To improve this performance, we developed a very simple technique to implement the virtual processors of the simulated system using light-weight threads rather than heavy-weight Unix processes. This approach was shown to improve performance by up to a factor of four on an iterative, synchronous scientific code.

We are now in the second phase of the project, where we are investigating approaches to augmenting LAPSE on a workstation cluster with temporal capabilities. Initially, we will determine if we can accomplish this goal with a very simple extension to a technique currently employed by LAPSE on the Intel Paragon. If this does not prove to be a viable option, we will investigate more sophisticated techniques.

References

[1] D. Agrawal, M. Choy, H.V. Leong, and A. Singh. Maya: A simulation platform for distributed shared memories. In *Proceedings of the 8^{th} Workshop on Parallel and Distributed Simulation*, pages

151–155, July 1994.

[2] E. A. Brewer, C. N. Dellarocas, A. Colbrook, and W. E. Weihl. Proteus: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.

[3] D.-K. Chen, H.-M. Su, and P.-C. Yew. The impact of synchronization and granularity on parallel systems. In *Int'l. Symp. on Computer Architecture*, pages 239–248, May 1990.

[4] x,x,x. Parallelized direct execution simulation of message-passing programs. *IEEE Transactions on Parallel and Distributed Systems*, Volume 7, Number 10, October, 1996, pages 1090-1105.

[5] F.W. Howell, R. Williams, and R.N. Ibbett. Hierarchical architecture design and simulation environment. In *MASCOTS '94, Proceedings of the Second International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*, pages 363–370, Durham, North Carolina, 1994. IEEE Computer Society Press.

[6] x,x,x,x. Towards a Thread-Based Parallel Direct-Execution Simulator In it Proceedings of the 29th Annual Hawaii International Conference on System Software, Maui, Hawaii, 1996.

[7] R. Konuru, J. Casas, R. Prouty, S. Otto, and J. Walpole. A user-level process package for PVM. In *Proceedings of Scalable High Performance Computing Conference*, 1994.

[8] D.M. Nicol. Parallel discrete-event simulation of FCFS stochastic queueing networks.

In *Proceedings ACM/SIGPLAN PPEALS 1988: Experiences with Applications, Languages and Systems*, pages 124–137. ACM Press, 1988.

[9] D.M. Nicol. The cost of conservative synchronization in parallel discrete-event simulations. *Journal of the ACM*, 40(2):304–333, April 1993.

[10] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vettering. *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, New York, 1988.

[11] S. Reinhardt, M. Hill, J. Larus, A. Lebeck, J. Lewis, and D. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference*, pages 48–60, Santa Clara, CA., May 1993.

[12] S. Reinhardt, B. Falsafi and D. Wood. Kernel Support for the Wisconsin Wind Tunnel. In *Proceedings of the Usenix Symposium on Microkernels and Other Kernel Architectures*, September 1993.

[13] G. Stellner, S. Lamberts and T. Ludwig NxLib - a parallel programming environment for workstations clusters. In *PARLE'94, Parallel Architectures and Languages Europe*, Athens, 1994.