Towards A Thread-Based Parallel Direct Execution Simulator*

Phillip Dickens[a] **Matthew Haines**[a] **Piyush Mehrotra**[a] **David Nicol**[b]

[a] *Institute For Computer Applications in Science and Engineering MS 132C, NASA Langley Research Center, Hampton VA. 23681 USA*

[b] *Department of Computer Science, The College of William and Mary Williamsburg, VA 23187, U.S.A*

Abstract

Parallel direct execution simulation is an important tool for performance and scalability analysis of large message passing parallel programs executing on top of a virtual computer. However, detailed simulation of message-passing codes requires a great deal of computation. We are therefore interested in pursuing implementation techniques which can decrease this cost. One idea is to implement the application virtual processes as lightweight threads rather than traditional Unix processes, reducing both on-processor communication costs and context-switching costs. In this paper we describe an initial implementation of a thread-based parallel direct execution simulator. We discuss the advantages of such an approach and present preliminary results that indicate a significant improvement over the process-based approach.

1 Introduction

Direct execution simulation is an important tool for performance and scalability analysis of large message passing parallel programs executing on top of a virtual parallel computer. In this approach, the application code is executed directly to determine its run-time behavior and any references to the virtual machine are trapped and handled by simulator constructs.

A detailed direct execution simulator for parallel programs offers the potential for accurate prediction of parallel program performance on large, possibly as-yet-unbuilt systems. The approach does require a great deal of computation, but is a good candidate for parallelization. Execution of the application processes is a clear source of parallelism; one easily envisions a system where the discrete-event virtual machine simulator resides on one processor while a pool of other processors host the directly executing application processes. However, this solution will perform poorly in situations where either the communication path to the simulator becomes a bottleneck or where the simulator execution itself is a bottleneck. Therefore, it is important to parallelize the virtual machine simulator as well as the application processes.

We have developed 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 an Intel Paragon and has been ported to a cluster of Sun workstations using the *nx-lib* message passing library [?], which provides the message passing functionality of a Paragon on a network of workstations.

One performance measure for a direct execution simulator is *slowdown*, which is defined as the time it takes the simulator to execute the application code on N processors divided by the execution time of the code running natively on N processors, and is thus a measure of the simulation overhead. LAPSE has reported slowdowns between 1.8 and 139 depending on the application code and the number of processors [?].

While the slowdowns reported by LAPSE are very reasonable when compared to other direct execution simulators, it would be useful to improve this performance. One approach to improving performance is to change the way the virtual processors of the simulated system are implemented. LAPSE assumes the application code is written such that there is one process per physical processor. Given this assumption, each physical processor in the simulated system will have its own address space, and thus it is natural to implement each simulated physical processor as a heavy-weight Unix process.

The slowdowns measured for LAPSE on a workstation cluster are generally greater than those measured for LAPSE executing on the Intel Paragon. This is in large part due to the higher communication costs of the former which uses Unix sockets as the communication mechanism. This is in contrast to the Intel

*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.

Paragon which has a high speed mesh interconnection network. Additionally, each node on the Intel Paragon has a dedicated communication co-processor.

Given the high communication costs on the network cluster, it can become impractical to execute large parallel direct execution simulations using LAPSE. One way to increase its performance is to use lightweight threads rather than heavyweight Unix processes for each simulated virtual processor. This reduces not only the context switching costs but also the costs of communicating among virtual processors executing on the same physical processor. However, using lightweight threads to support virtual processors brings up a very important issue: how do we provide the separate address spaces for each of the simulated virtual processors when threads are generally implemented such that they all share one global address space?

In this paper we discuss our approach to providing lightweight threads with a separate address space and then using these threads for implementing the virtual processors. We then compare the thread-based approach to that of the original process-based version of LAPSE. Our initial results suggests the thread-based approach can significantly improve the performance of LAPSE making it a practical tool for large parallel direct execution simulations.

Currently, the thread-based version of LAPSE is implemented only on a cluster of workstations. However, it should be quite simple to use our approach to implement the thread-based version of LAPSE on the Intel Paragon.

It is important to note that LAPSE is not the only direct execution simulator that employs light-weigth threads to improve performance. At least three other simulators use this approach (the Wisconsin Wind Tunnel project [?], Tango Lite [?] and the simulator [?] developed as part of the RPPT project [?]). What makes our work unique is that we provide threads with separate address spaces without special operating system support and without requiring that all private data be maintained on the stack of a particular function of a thread. These issues are discussed more fully in section 5.

The remainder of the paper is organized as follows. In Section 2 we briefly review parallel direct execution simulation and give some background information on LAPSE. In Section 3 we discuss our implementation of the process-based LAPSE and discuss the performance implications of this approach. In Section 4 we give an overview of thread-based computation and in Section 5 we detail our approach for providing separate address spaces among threads. In Section 6 we present preliminary experimental results comparing the process-based and thread-based versions of LAPSE executing on a workstation cluster. We conclude and provide future directions in Section 7.

2 Background

Scalability is an important area of current research in parallel processing. A programmer is interested in how, given a particular parallel algorithm and architecture, a program will perform as the problem size and the number of processors increases. A code that achieves good performance on a small number of processors may not achieve similar performance gains on a large number of processors. Users are interested in knowing if and why this happens. One may also wish to know how the application will perform under different releases of the operating system (with different overheads) or different network architectures. The most direct way of answering these questions is to execute the code using many processors on the software and hardware of interest. However, this isn't always practical. Massively parallel machines are expensive and are shared among many users. It can be difficult, expensive, or infeasible for a user to acquire the full resources of a massively parallel machine on a regular basis. However, smaller numbers of processors of such a machine may be routinely available. This small number of processors can then be used to emulate the code running on the larger system, and to predict the time it would have taken to execute the application on the larger system. Similarly, the code of the parallel multicomputer could be emulated on a network of interconnected workstations. Additionally, designers of large distributed systems could make use of a parallelized simulation of the network, driven by executing application code, as could designers of new communication networks.

Direct execution simulation is a mechanism to predict the behavior for some *virtual computer* of interest using some other physical machine [?, ?, ?, ?]. Given N application processes whose performance on N processors is sought, we use $n < N$ processors to both execute the application and simulate its timing behavior. Each physical processor is assigned some number of application processes (virtual processors, or VPs) and a 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 handled by simulator constructs. The simulator processes determine how simulation time advances as a function of the actual application process execution and the simulated machine behavior. This approach promises fairly rapid evaluation of a code's scalability and the possibility of monitoring network behavior—albeit simulated behavior—in a way that is not normally possible on actual codes.

LAPSE provides performance and scalability predictions of large message-passing application codes written for the Intel Paragon. In particular, LAPSE allows a user to predict the performance of an application code executing on a large virtual Intel Paragon using a much smaller physical machine. As noted before, another version of LAPSE allows a network cluster to be used to obtain performance and scalability analysis for codes written for the Intel Paragon.

Several other projects use direct execution of application processes to drive simulations of multiprocessor systems [?, ?, ?, ?, ?]. The Wisconsin Wind Tunnel (WWT) [?] 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, being designed to simulate a different type of machine than its host. 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 [?] while LAPSE uses a synchronization mechanism combining YAWNS with appointments [?]. We do not elaborate on the details of the synchronization mechanism here. The interested reader is directed to [?] for a detailed discussion. The third difference is in the implementation of virtual processors using light-weight threads. As will be discussed, the approach taken in [?] requires significant operating system support while our approach does not.

3 LAPSE

In this section we provide a brief discussion of the structure and use of LAPSE. These issues are discussed more fully in [?].

To use LAPSE, a programmer simply modifies the application code's makefile to call LAPSE scripts instead of native compilers, and sets up a file specifying LAPSE parameters, such as the number of simulated and the number of actual nodes. The LAPSE system transforms copies of the code's files (which may be in C, or Fortran, or both) into a set of "application" and "simulation" processes. In the process-based version each actual node multi-tasks one simulator process and multiple application processes. For every node in the simulated system, LAPSE creates a unique application process that executes functionally exactly the way the application code would in the simulated system However, wherever the original application makes a call to the simulated virtual machine (e.g., a send, receive, probe, or clock call), the LAPSE modified application calls a LAPSE routine to perform that action, and to report it to the simulator process. The net effect is that the application processes communicate exactly the same messages as they do in the simulated larger system, but the LAPSE simulation processes simulate the functionality of the simulated larger system. In doing so, the LAPSE simulators provide temporal information about the simulated system. For instance, whenever the application process calls a system clock (as it will when performance tuning), LAPSE will return to the application the time in the *simulated* system.

To help capture temporal information LAPSE gets execution time information from application processes. At compile-time LAPSE modifies the application assembly code, inserting instruction counters at basic block boundaries. Whenever an application process calls a LAPSE routine, LAPSE recovers the number of instructions executed since the last call to a LAPSE routine. This provides us with a measure of the time spent executing the application in the simulated system. This measure, along with a description of the requested activity, is sent to the LAPSE simulation process responsible for this application process. The simulation processes are distributed across the actual system, one per node, each one being responsible for simulating the timing behavior of all application processes also assigned to that node. As noted above, synchronization between simulation processes is required since the simulation in one process can affect and be affected by activity in other simulation processes. However, one of the key insights identified by our work is that in many codes of practical interest, LAPSE enjoys excellent "lookahead" capabilities since the application processes can be run largely as on-line trace-generators with little synchronization with the simulation processes [?].

3.1 LAPSE on a workstation cluster

As noted before, we have ported LAPSE to a workstation cluster using *nx-lib* [?]. *nx-lib* is a software library developed by researchers at the University of Munich allowing the development and execution of codes for the Intel Paragon multicomputer, using an ordinary network of workstations. Codes run under *nx-lib* have the functionality of Paragon codes. However, owing to temporal sensitivities, it is possible for the execution path of a code to be different on the workstations than it would be on the actual 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 be used to provide accurate timing estimates of how long the code would have taken had it been run on the Paragon.

There are two primary issues which must be addressed when porting LAPSE to the workstation cluster. The first issue is how to make performance predictions for a code developed for an Intel Paragon when it is actually executing on a workstation cluster. The second issue is how to improve the performance of application codes running under LAPSE. In this paper we are concerned primarily with increasing the performance of direct execution simulations executing under LAPSE, the former is the focus of current research.

The implementation of *nx-lib* on the workstation cluster assigns one TCP socket for each application process in the system. As noted, LAPSE creates one Unix process for each simulation and application node. LAPSE can be communication intensive since each of the application processes interact with each other as well as with the simulation processes. Additionally, the simulation processes frequently communicate with each other. Since each such communication uses a Unix socket, the cost of both on-processor and off-processor communication is very high.

Since communication costs are so critical to LAPSE executing on a workstation cluster it is important to explore ways to minimize this cost. One approach is to implement the application processes as lightweight threads rather than heavyweight Unix processes. This provides two benefits: First, it reduces on-processor communication costs by replacing TCP sockets with memory copies, and second, it reduces the cost of context switching. For these reasons implementing application processes as lightweight threads is very attractive.

Implementing application processes as lightweight threads requires some mechanism to provide each thread with its own address space. This is because each application process is written just as it would be if it were to execute on a physical processor of the Intel Paragon. There is no shared memory on the Intel Paragon and thus each process accesses its local and global variables without concern for interference from another process. How to provide this distributed memory environment using a lightweight threads package is briefly discussed in the section 5.

4 Thread-based computation

Lightweight threads are becoming a popular mechanism for expressing asynchronous behavior and potential parallelism for many current languages and systems [?, ?, ?]. In response to this demand for threaded systems, many lightweight thread packages have been developed [?, ?, ?, ?, ?]. Additionally, the POSIX committee has established a standard interface for lightweight threads called *pthreads* [?], and a library implementation of this proposed standard already exists [?].

4.1 Definition

A *thread* represents an independent, sequential unit of computation that executes within the context of a kernel-supported entity, such as a Unix process. Threads are often classified by their "weight," which corresponds to the amount of context that must be saved when a thread is removed from the processor, and restored when a thread is reinstated on a processor (i.e. a *context switch*). For example, the context of a Unix process includes the hardware registers, kernel stack, user-level stack, interrupt vectors, page tables, and more [?]. The time required to switch this large context is typically on the order of thousands of microseconds, and therefore a Unix processes represents a *heavyweight* thread. Contemporary operating system kernels, such as Mach, decouple the thread of control from the address space, allowing for multiple threads within a single address space and reducing the context of a thread. However, the context of a thread and all thread operations are still controlled by the kernel, which must often include more state than a particular application cares about. Context switching times for kernel-level threads are typically in the hundreds of microseconds, resulting in a medium or *middleweight* thread. By exposing all of the thread state and operations at the user-level, a minimal context for a particular application can be defined, and operations to manipulate threads can avoid crossing the kernel interface. As a result, user-level threads can be switched in the order of tens of microseconds, and are thus termed *lightweight*. For the remainder of this paper, we will use the terms "thread," "lightweight thread," and "user-level thread" synonymously.

4.2 Computation model

The execution model for lightweight threads is very similar to the execution model of a process within the Unix operating system. All threads are executed within the context of a process, and the thread-level execution is controlled by a thread-level *scheduler*, whose job is to determine the next available thread that should execute and to switch between threads at context-switching points. Thread scheduling can either be

- *non-preemptive* (typically FIFO), in which a thread will continue to execute until it completes, explicitly yields control of the processor, or blocks on a synchronization primitive; or
- *preemptive* (typically round-robin), in which each thread is given a time quantum and is interrupted when the quantum expires.

It is important to note that all threads execute within a process and are therefore subject to the process-level scheduling, which is typically preemptive round-robin. When a process is suspended, so too are all threads within that process.

5 Providing separate address spaces for threads

In Unix each process is assigned a separate address space whose boundaries are enforced by the operating system. This is done so that the operating system can interleave the execution of user programs without the possibility of one process corrupting the address space of of another. This is accomplished by indirectly routing all memory references through a *base address register* which is different for each process, and updated by the operating system when processes are restored [?].

Lightweight thread packages define and execute threads within the context of a single Unix process, hence a single address space. Therefore, variables defined outside of the scope of a thread function (file scope) are shared by all threads, and in fact are allocated in a single, global block of memory. This is typically beneficial as it allows threads to easily share information using common addresses combined with mutual exclusion operations to ensure consistency. However, many applications, such as LAPSE, require that a certain amount of context be maintained on a per-thread basis across procedure calls. This type of information is referred to as *thread-specific* data. The issue for LAPSE is that when emulating virtual distributed processors using threads, it requires data that is private to a thread but global with respect to all functions within the thread.

5.1 Approaches

Most lightweight thread systems provide some mechanism for dealing with a small amount thread-specific data. For example, pthreads [?] provides a general purpose key/value mechanism that allows a user to create a thread-specific data key that is shared among all threads, where the value of the key can be set differently for each thread. This approach is designed for managing a small amount of thread-specific data, such as a copy of `errno` for each thread, but is not well-suited for actually providing thread-specific separate address space within a process.

This is the approach used by Tango Lite [?] when separate address spaces for threads are required. Tango Lite is used primarily to simulate shared memory architectures and it is therefore natural to implement the virtual processors as threads which all operate within the same global address space. If separate address spaces are needed, the private data must be declared locally to each thread. While this does give each thread its own copy of all variables it does not give each function within the thread access to those variables (since the variables are declared and stored on the stack of a particular function). This approach cannot be used in LAPSE because the variables must be private to each thread while being accessible to all functions within the thread.

Another approach to providing thread-specific address spaces is to access and modify the base address register that the operating system uses to support separate address spaces for each process. This approach was used to provide a virtual process implementation of PVM [?], but suffers from two main problems. First, many processor architectures do not provide access to the base address register, so this approach is not portable among machines. Second, special save/restore code is required to manage the base address register, so this approach is also not portable among lightweight thread systems.

The Wiscon Wind Tunnel project [?], also depends upon significant support from the operating system to provide separate address spaces for threads. In particular, it requires that the operating system allow WWT to create subservient contexts (address spaces), to manipulate the page mappings within each subcontext, to handle traps generated during the execution of a subcontext and to manipulate memory tags in subcontexts [?]. We wanted to keep LAPSE portable and hence had to find an approach that would require no special support from the operating system.

Finally, the thread-specific data mechanism offered by most thread packages can be used to maintain a base address pointer which is different for each thread. The idea is that each thread allocate a separate copy of each global variable and access each global variable indirectly through this base address pointer. While this provides a machine and system-independent solution, it requires significant modification to the user code so that all global variables are referenced through the base address register rather than being accessed directly. We know of no project which uses this approach.

5.2 Our solution

Our solution is to employ 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, and without significantly affecting the performance of a context switch.

Our solution is very similar to the approach of maintaining a separate base address pointer through which all global data is referenced. By correctly saving and restoring this variable for each thread, we can effectively create separate address spaces. However, rather than attempting to re-write the user code to add the indirection for all global data, we take advantage of the C++ class scoping mechanism to provide this capability with minor modifications to the user code.

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 1.

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 that are 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 proceesor, using the thread entry function shown above. At this point each thread has its own "private" address space via the class **G_Class**.

One drawback of our approach is that it requires a C++ compiler to provide the proper scoping, and therefore interfacing with Fortran user code could pose some problems. We present the results of our threaded-LAPSE implementation in the next section.

```
# 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 1: Creating Separate Address Spaces

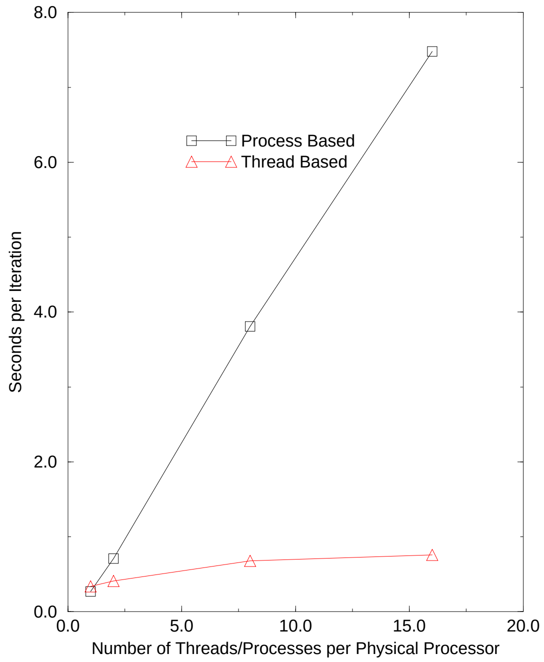

Figure 2: High Communication/Computation Ratio Using 16 Virtual Processors

6 Experimental results

One of the primary expected benefits of a thread-based execution is much lower communication costs due to a reduction of the interprocess communication time when threads instead of processes are used to simulate virtual processors. To get a measure of this improvement we present a set of experiments which compare the two approaches using a communication intensive application. Also, we present a set of experiments which capture the impact on performance due to the communication/computation ratio.

/* The application code employs a constant 16 virtual processors, and we simulate the code using 1,2,8 and 16 physical processors (thus having 16, 8, 2, 1 processes (threads) per physical processor). Additionally, each physical processor executes one simulator process (at the time of this writing we have not integrated the simulator process into our thread based approach). The experiments were conducted using a cluster of Sun Sparcstation 20s. Before presenting our results it would be useful to briefly discuss our implementation of message passing between virtual processors.

As noted above, the process-based version of LAPSE uses Unix sockets for all communication regardless of whether the processes reside on the same or different physical processors. In the thread-based version, for each communication it must be determined whether the message is for an on processor thread or an off processor thread. If the thread is on processor a simple memory copy is used to transfer the data. If it is off processor, Unix sockets are used to transfer the data.

The application code goes through several iterations of a compute/communication cycle. In the compute phase, the application performs some number of integer operations, and in the communication phase the application first sends and then receives a message from its nearest neighbor in a ring configuration. Our normalized metric of performance is the number of seconds required to complete one compute/send/receive iteration.

We vary the communication/computation ratio by varying the number of integer operations in the compute phase of the cycle. In the first experiment the application does no computation (high communication/computation ratio), in the second experiment the application performs 10,000 integer operations (medium communication/computation ratio), and in the final experiment the application performs 250,000 integer operations (low communication/computation ratio). The results for the high communication/computation ration are shown in Figure ??.

As can be seen, the amount of time required to complete one iteration of the algorithm for the process-based version increases very quickly as the number of processes per physical processor is increased. The communication costs of the thread-based approach increase at a much lower rate. When the number of processes per physical processor is 16 the process-based approach requires approximately 10 times as long to

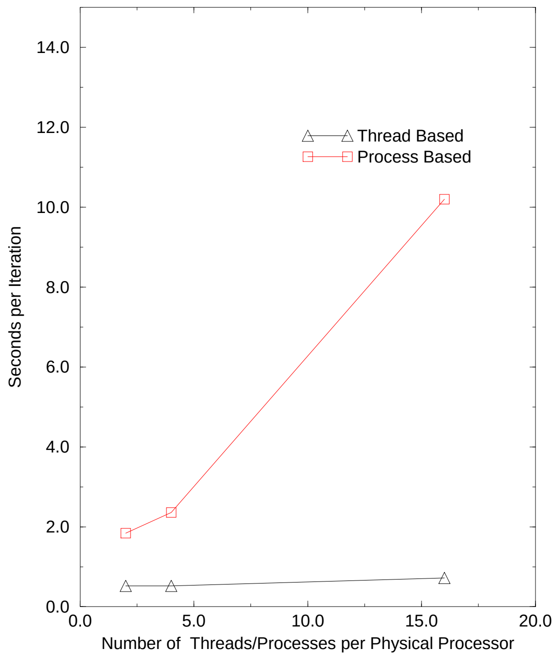

Figure 3: Medium Communication/Computation Ratio for 16 Virtual Processors

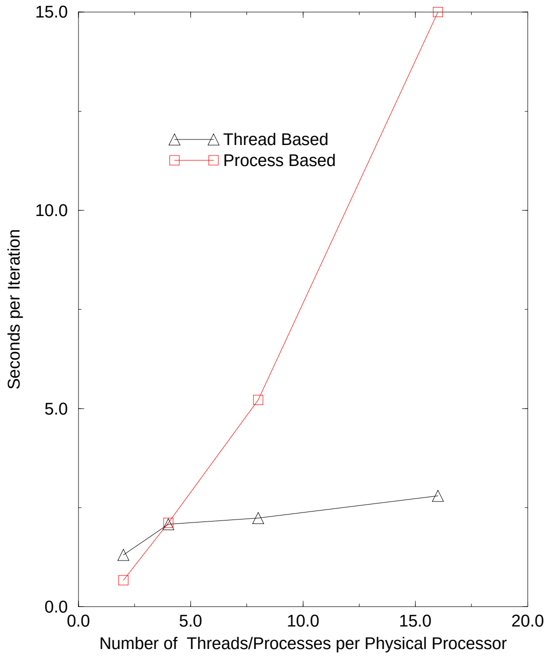

Figure 4: Low Communication/Computation Ratio for 16 Virtual Processors

execute one iteration of the algorithm. This experiment confirms that communication costs are significantly lower using a thread-based approach.

Note that the process-based approach is a little bit faster than the thread-based approach when there is one process/thread per processor, which can be attributed to the overhead of the lightweight thread system. In particular, using threads requires routines and data structures to determine such things as whether the recipient of a particular message resides on/off processor, and whether a particular message buffer is free. When there is only one thread on a given processor this unnecessary machinery can be avoided which would increase performance. We have not yet optimized for the one thread per process case.

As shown in Figure ??, the thread-based approach still significantly outperforms the process-based version for the medium communication/computation ratio. Again the costs for the thread-based approach rise much more slowly than for the process-based approach.

Figure ?? shows the improvement in performance when the communication/computation ratio is low. Again it is seen that the costs associated with the process-based approach grow much more quickly than the thread-based approach as the number of processes (threads) per processor is increased. It is interesting to note that the process-based approach slightly out-performs the thread-based approach for the two and four processes (threads) per processor. This most likely has to do with thread scheduling as we now discuss.

Our preliminary implementation of the thread-based approach is non-pre-emptive, and thus the threads execute until they have no more useful work to perform. One example of where a thread would give up control of the processor is when it issues a call to receive a message and the message is not available. In our experiments, each thread requests a message during each iteration. Since there is no guaranteed order of execution it is unknown whether a particular message will be available for a particular thread when it is required. Thus the scheduler may have to swap several threads in and out before finding one eligible for execution. The process-based approach is time-sliced, making it more likely that each process will be progressing at roughly the same rate which may mean fewer context switches to find a process eligible for execution. This issue of thread scheduling merits further investigation.

All of the experiments reported here make it clear that thread-based implementations offers the possibility of significant performance improvement. Also, our studies show that this improvement in performance is not very sensitive to the the communication/computation ratio. Clearly we need to develop a test suite for real direct execution simulation applications, but our preliminary results are very encouraging.

Before leaving this section it is important to note that these results do not address the important issue of relative speedup due to parallelization of the virtual computer simulator. This issue is discussed in detail in [?].

7 Conclusions

This paper outlines the idea of using lightweight threads to support a parallel direct execution simulator, such as LAPSE. The traditional process-based implementation suffers from high context-switch and interprocess communication overheads, and we show that a thread-based implementation can reduce much of this overhead. In support of this approach, we have outlined a method for providing thread-specific address spaces, something not commonly supported by lightweight thread systems but required for supporting virtual processors.

Preliminary results show that the thread-based LAPSE offers the potential for significant performance improvement over the process-based approach. We observed up to a ten fold improvement for the ring based message-passing code.

Based on the results of the research presented here we feel that a thread-based approach to direct execution simulation bears further investigation. Our next step is to modify existing application codes to run on the thread-based LAPSE simulator. This would allow us to more carefully investigate issues such as the effect on performance due to the computation/communication ratio and varying communication patterns.

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] Maurice J. Bach. *The Design of the UNIX Operating System.* Software Series. Prentice-Hall, 1986.

[3] Henri E. Bal, M. Frans Kaashoek, and Andrew S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Transactions on Software Engineering*, 18(3):190–205, March 1992.

[4] Brian N. Bershad. The PRESTO user's manual. Technical Report 88-01-04, Department of Computer Science, University of Washington, January 1988.

[5] 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.

[6] 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.

[7] R. Covington, S. Dwarkadas, J. Jump, S. Madala, and J. Sinclair. Efficient simulation of parallel computer systems. *International Journal on Computer Simulation*, 1(1):31–58, June 1991.

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

[9] P.M. Dickens, P. Heidelberger, and D.M. Nicol. Parallelized direct execution simulation of message-passing programs. Technical Report 94-50, ICASE, July 1994.

[10] S. Dwarkadas, J. Jump, and J. Sinclair. Execution-Driven Simulation of Multiprocessors. InACM Transaction on Modeling and Computer Simulation, Volume 4 Number 4, Pages314-338.

[11] I. T. Foster and K. M. Chandy. Fortran M: A language for modular parallel programming. Technical Report MCS-P327-0992 Revision 1, Mathematics and Computer Science Division, Argonne National Laboratory, June 1993.

[12] R. M. Fujimoto. Simon: A simulator of multicomputer networks. Technical Report UCB/CSD 83/137, ERL, University of California, Berkeley, 1983.

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

[14] Dirk Grunwald. A users guide to AWESIME: An object oriented parallel programming and simulation system. Technical Report CU-CS-552-91, Department of Computer Science, University of Colorado at Boulder, November 1991.

[15] Carl Hauser, Christian Jacobi, Marvin Theimer, Brent Welch, and Mark Weiser. Ising threads in interactive systems: A case study. In *ACM Symposium on Operating System Principles*, pages 94–105, Asheville, NC, December 1993.

[16] S. Herrod. Tango Lite: A Multiprocessor Simulation Environment. Unpublished Introduction and User's Guide.

[17] 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.

[18] IEEE. *Threads Extension for Portable Operating Systems (Draft 7)*, February 1992.

[19] David Keppel. Tools and techniques for building fast portable threads packages. Technical Report UWCSE 93-05-06, University of Washington, 1993.

[20] 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.

[21] I. Mathieson and R. Francis. A dynamic trace-driven simulator for evaluating parallelism. In *Proceedings of the 21st Hawaii International Confernce on System Sciences*, pages 158-166, January 1988.

[22] Piyush Mehrotra and Matthew Haines. An overview of the Opus language and runtime system. In *Proceedings of the 7th Annual Workshop on Languages and Compilers for Parallel Computers*, pages 346–360, New York, November 1994. Springer-Verlag Lecture Notes in Computer Science, 892. Also Appears as ICASE Technical Report 94-39.

[23] Frank Mueller. A library implementation of POSIX threads under UNIX. In *Winter USENIX*, pages 29–41, San Diego, CA, January 1993.

[24] Bodhisattwa Mukherjee, Greg Eisenhauer, and Kaushik Ghosh. A machine independent interface for lightweight threads. Technical Report CIT-CC-93/53, College of Computing, Georgia Institute of Technology, Atlanta, Georgia, 1993.

[25] D. Nicol, C. Micheal, and P. Inouye. Efficient aggregation of multiple LP's in distributed memory parallel simulations. In *Proceedings of the 1989 Winter Simulation Conference*, pages 680–685, Washington, D.C., December 1989.

[26] 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.

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

[28] 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.

[29] 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.

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

[31] Carl Schmidtmann, Michael Tao, and Steven Watt. Design and implementation of a multithreaded Xlib. In *Winter USENIX*, pages 193–203, San Diego, CA, January 1993.

[32] Sun Microsystems, Inc. *Lightweight Process Library*, sun release 4.1 edition, January 1990.

[33] 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.