Final Technical Report

Hardware Supported Multi–Core Communications for Efficient Parallel Discrete Event Simulations

Prepared by: Dr. George F. Riley
Georgia Institute of Technology

Introduction

It is well known that there is significant overhead in existing methods for executing a parallel or distributed discrete event simulation (*PDES*). This overhead is due to two different requirements, namely the overhead for *time synchronization* and the overhead for *message passing*. In this context, time synchronization refers to the requirement for any conservative parallel discrete simulation to continually be aware of and compute the *lower bound on timestamp* (*LBTS*), which is the global minimum of the timestamp for all unprocessed messages in the simulation. Traditional methods for computing the *LBTS* value require $O(NlgN)$ message exchanges between processors, where N is the number of processors executing the *PDES*. Message passing is required by *PDES* applications due to the fact that an event generated by processor A might in fact affect simulated objects modeled on a different processor B. Anytime this occurs, the contents of the event must be serialized and copied, usually across address space boundaries.

With the present and future proliferation of Multi–Core architecture computing devices, we expect that parallel programming, and parallel discrete event simulation in particular, will become more prevalent. In research presented here, we explored ways to reduce dramatically the overhead described above for *PDES* applications by designing and testing specialized hardware that will be incorporated into multi–core architecture designs. Once this hardware is incorporated into multi–core architectures, the overhead and message passing will be reduced to near zero, independent of the number of cores in the architecture.

Background

In prior work over the past two year, we have designed and evaluated on a small scale a novel hardware–supported time–synchronization unit we call the *Global Synchronization Unit*. This novel device allows individual cores in a multi–core enviornment to determine a global minimum timestamp, ane *accounting for transient messages* in just a few clock ticks. We have demonstrated the viability and effectiveness of the approach using instruction–level simulation tools on up to eight cores. In the work reported here, we extended that prior work on the *GSU* and evauluated it on up to 32 cores. Secondly, we designed a second set of specialized devices that allow *zero–copy message passing* between logical processes (cores) in a shared–memory distributed simulation. It is well known that message passing overhead can be the most significant bottleneck in shared–memory distributed simulations. Our *zero–copy message passing* (*ZCM*) approach that allows the various CPUs in a multi–core architecture to send and receive messages without interlocks and without expensive message copying operations. Our design uses a shared memory region and a series of circular mailbox queues (in hardware), along with specialized hardware to allocate and free messages in the shared memory region.

Overview of the Global Synchronization Unit

As mentioned previously, a conservative *PDES* application requires frequent computation of the *lower bound on timestamp* value. Current implementations accomplish this computation by exchanging messages between simulator instances, which can result in excessive overhead as the number of processors grows. Another approach, discussed by Carothers [1], uses specialized interconnect hardware in high–end supercomputers to gather this global consensus information. Our approach differs from that work since we target low–end commodity multi–core chips, rather than multi–million dollar supercomputing platforms.

In prior work, we have designed and demonstrated the efficiency of our *Global Synchronization Unit* (*GSU*). The *GSU* consists of several register files, each with with N values, where N is the number of CPUs in the multi–core architecture. One register file is called the *Minimum Outstanding Event* (*MOE*) register. At any point in time, each CPU executing a *PDES* will insure that the timestamp of the smallest unprocessed event in its local event queue is written in its corresponding entry in the *MOE*. Then, computing the minimum value is just a set of $N - 1$ comparators in lgN ranks to find the minimum value. However, this simple approach does not take into account the possibility of *transient messages* between any two CPUs in the *PDES*. To account for this, the design includes two additional register files (also N deep). The *Minimum Outstanding Message* (*MOM*) register file contains the timestamp of the smallest transient message destined for each process, and the *Transient Message Count* (*TMC*) register file contains the count of transient messages. Our design includes specialized atomic access instructions to insure that race conditions are properly handled. With these two additional register files, the overall minimum value (the actual *LBTS*) is then just the minimum of the two minima in the *MOE* and *MOM*. We have shown that this design correctly reports the true *LBTS* value, that this results in a near zero overhead lookup of *LBTS* and the resulting *PDES* applications exhibit significant speedup. Details of this design and performance analysis can be found in [2].

Overview of Hardware-Supported Zero–Copy Message Passing

Another source of significant overhead in a typical *PDES* application is in the message passing between the processes. Even in a tightly coupled, shared memory environment, inter–process messages must be copied to a shared memory region. After the message is copied, the message recipient must be somehow informed that the message exists and where it is located in the shared memory region.

We designed an approach that allows interlock–free and zero–copy message passing in a multi–core parallel application. The design uses new atomic instructions to read and write a circular message queue at each CPU, and hardware–supported shared memory management. Rather than allocating memory from a process–private heap and copying messages to a shared memory region, our approach manages the shared memory heap directly, allowing processors to directly allocate, free, and pass offset pointers to the shared memory region. Hardware *usage count* registers automatically manage the memory in the shared heap, allowing efficient zero–copy message passing between processes.

Using this approach, to send a message between processors, a process simply requests an atomic allocation of consecutive memory locations in the shared memory region. It then uses another atomic instruction ("send message") to write the offset of the allocated memory (relative to the start of the shared memory heap) into a hardware circular queue mailbox at the receiving processor. Our design allows for the case where the receiving mailbox is full, and will allow the sender to specify a number of clock cycles to wait for space, after which the instruction returns a failure flag. In this case, the sender simply re–tries sending the message immediately, or at a later time.

Software–Supported Zero–Copy Message Passing

The work reported above in the design of a hardware–supported zero–copy message passing approach led us to realize that we could achieve almost as good performance with a software-only approach to zero-copy message passing. Utilizing the well–known shared memory allocation methods found in all *Linux* platforms, and coupling that with well–known reference counted smart pointers, we could design a working version of the zero-copy approach without any specialized hardware. Since this work consumed the majority of our efforts during the period of performance, we give a very detailed discussion of the approach and the results of our extensive performance analysis experiments below.

Introduction

The increasing demand for longer and more detailed simulations of complex systems has led to a rise in parallel discrete event simulation (PDES). Parallelization allows these applications to take advantage of the current architectural trend of stamping multiple CPUs (cores) on to a single die. With two to six cores per chip now commonplace, and with eight to sixteen cores per chip planned in the near future, simulations of large complex systems can occur on multiple CPUs in a tightly coupled environment.

There are two main methods available to take advantage of the multiple cores on today's CPUs. The first is the thread model in which each logical process (LP) runs on its own thread within one operating system process. With multithreaded applications, data can easily be passed between concurrently running threads using simple C type pointers. This allows the threads to communicate using messages of any size at the cost of passing a 32 or 64 bit pointer.

The alternative is the multi-process model where each logical process runs in its own individual operating system process. With multi-process applications, the process of passing messages becomes more complicated. The operating system runs each individual process in its own virtual address space, and any pointers created in a process will reference a virtual address, not a physical address. Since the mappings between virtual and physical addresses might be different for each process, pointers created by one process and passed to another might refer to a completely different physical location in memory for the receiver. Therefore, standard C type pointers can not be used as a means to pass messages between individual processes.

Both the multi-threaded and multi-process distributed simulation approaches are commonly used.

In general, a distributed simulation using multiple threads will need all event handlers and the event scheduling engine to be aware of the need for multiple-access interlocking to prevent simultaneous updates and potential deadlocks. In contrast, when distributing the simulation in separate address spaces, only those portions of the simulation that send events and the portion that advances simulation time needs to be aware of the distributed execution. Further, if the original simulation package was not designed with distributed execution in mind, the multi-process approach is in general considerably easier to implement. *ns–2* is an example of simulation environment that was not in fact designed originally to execute in a distributed fashion, but was later adapted to execute with multiple processes and disjointed address spaces.

A common API to use when working with multi-process simulations is the well–known *Message Passing Interface* (*MPI*). However, a significant overhead for MPI based applications is in message passing between the disjointed address spaces. Even in a tightly coupled, shared memory environment, the messages must be copied into a shared memory region. The copying of the entire message must be done because simple, C type pointers use virtual addresses which have no meaning once the pointer is passed to another process. Thus a common approach is to first serialize the message and its data and perform a memory copy of the serialized message to a shared memory location, where it then can be retrieved and de-serialized by the receiving process. While this technique works and is in common use, a large processing cost can be incurred due to the amount of data being copied between processes. In the cases where these messages are large and must be passed frequently, this becomes a significant limiting factor in overall application performance.

An alternative approach is to use shared memory and smart pointers. With this method, data that needs to be shared with other processes can be created in a shared memory region that is accessible by every process. Then when a message needs to be passed, the owner can pass to the receiver a specialized smart pointer. These smart pointers allow the processes to pass only metadata for messages being exchanged, rather than the complete message. The receiving process can then access the original copy of the data stored in the shared memory using the smart pointer and normal dereferencing semantics. This greatly reduces the amount of data that needs to be passed between applications. This technique is referred to as Zero-Copy message passing.

We are not the first to develop a Zero-Copy approach. Boost Interprocess offers a smart pointer design that allows shared memory usage. Also provided by Boost is a general purpose, shared memory allocator. However, as we will show, the implementation provided by Boost does not perform well in memory intensive applications, like distributed simulators, and also does not scale well to a larger number of LPs. In contrast, our implementation of Zero-Copy is designed to function on a large number of LPs in a memory demanding environment. Along with the other benefits of Zero-Copy, our design supports *Global Virtual Time* and will not re-assign any freed memory area until the *GVT* equals or exceeds the *Simulation Time* when the final reference has freed the region. Thus, this technique can be adapted to work with optimistic synchronization algorithms with rollbacks. Using this new approach, performance of PDES applications on multi-core architectures is greatly improved, allowing for longer and more detailed simulations.

Background and Motivation

In this section we provide background material on message passing and time synchronization algorithms. In general, Zero-Copy message passing offers many benefits over standard shared memory message passing. Thus, to fully appreciate these benefits, understanding of the mechanics of message passing is necessary. Next, we provide a brief background on time synchronization algorithms. Our Zero-Copy implementation works with the two major classifications of time synchronization algorithms, and both will be discussed, so a brief background is provided for reference.

Traditional Message Passing

Commonly, messages have been passed between individual processes by copying the entire content of the message to a shared–memory region accessible by both the sender and the receiver. However the data to be transferred must first be formatted in a way that allows it to be meaningful to a receiver. This formatting, called serialization, or marshalling, generally copies each individual data item for all messages (and dereferening pointers as needed) to a sequential array of bytes, which are then copied to the shared memory region. In some cases, it is possible to marshall the data directly to the shared memory region, eliminating one of the memory copies needed. However this is rarely a trivial process because complex data objects generally make extensive use of pointers, references to other locations in memory. The data referenced from these pointers must also be copied into the buffer because, as mentioned previously, the pointer will no longer be valid once the data is transferred to a different process. Thus the serialization process, in most cases, involves numerous and often recursive memory copies.

Once the data to be sent has been serialized, it can then be transferred to the receiver. However, before the receiver can make use of what it has received, it must first deserialize, or un–marshall, the data. This is basically a reversal of the serialization process and therefore frequently requires multiple memory allocations to restructure complex objects back into their original form.

This process is depicted in Figure 1. While this procedure works as intended, the time to complete this marshalling and copying process is non-negligible and increases linearly with the size and complexity of the message.

Introduction to Zero-Copy Message Passing

Zero Copy message passing improves the performance of parallel simulations in a multi-process environment utilizing two main components: shared memory and smart pointers. First, rather than copying the data to shared memory, the data is created in shared memory, which can be accessed directly by the receiving process. As standard C type pointers cannot be utilized in this type of environment, Zero-Copy utilizes a smart pointer which is aware of the shared memory nature of the underlying data, and which has normal pointer semantics for dereferencing the pointer to access individual data items. Using the smart pointer, eliminates the need for performing an expensive memory copy. Furthermore, the smart pointer can be equipped with reference counting semantics, which results in the underlying shared memory area being *freed* when all smart pointers pointing

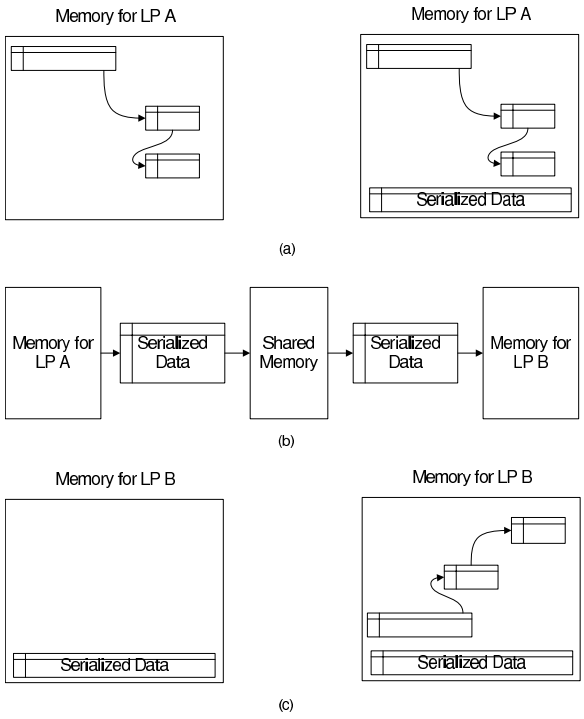

Figure 1: The transfer of a complex object between logical processes using a memory copy. (a) The complex object is serialized. (b) The serialized data is first copied to shared memory and then copied into the memory space of LP B. (c) The data is finally de-serialized and the object is ready to be used.

to the same area have gone out of scope. In this section we will describe our implementation of Zero-Copy and compare it to the implementation found in the Boost Interprocess C++ Library.

Boost Interprocess C++ Library

Boost provides an extensive library for working with shared memory [3]. These classes can greatly simplify the task of working with shared memory. Here we will discuss the Boost offset pointer and the Boost managed memory segment classes, which when used together, can form a Zero-Copy implementation.

In Boost, the smart pointer that works with shared memory is the *offset pointer*. The offset pointer stores the distance from the offset pointer's address to the object the pointer refers to. This allows objects created in shared memory to refer to each other regardless of which base address the shared memory segment is mapped into the processe's address space. To the programmer, the pointer functions equivalently to a normal pointer and can be coupled with a reference counting pointer to supply automatic garbage collection. The problem with this design is that for it to work correctly, all of the objects have to be stored in the same shared memory segment. There is no guarantee that the difference in base addresses of two shared memory segments mapped into the address space of a process will be the same between processes. Therefore, in most cases, only *one* shared memory segment can be used for all LPs.

A pool of shared memory is obtained in Boost using *managed memory segments*. Once a pool is available, objects can be created using the segment's allocator. The segments provide an *allocate()* method that takes as a parameter a byte size and returns a void pointer to a chunk of memory that size if it is available in the segment. The segments also provide a templated *construct()* method which will create an instance of the object specified in shared memory and return a pointer to it. Managed memory segments in Boost also allow the programmer to create named shared memory objects. A string name can be given to any object created in shared memory. This name can then be used by any process that connects to the memory segment to find the object in shared memory. While the Boost shared memory segment offers many useful features, it is unable to perform satisfactorily in a memory intensive environment, as we will show. In addition, as mentioned above, due to how Boost's smart pointer interacts with memory segments, only one such segment can typically be used in an application. This has two major consequences. First, access to critical sections of code in the segment's allocator can become a major bottleneck as the number of LPs increase. Second, if the memory in the one shared memory segment is exhausted, the application has no choice but to terminate since shared memory segments are not dynamically expandable.

Shared Heap

In our Zero-Copy message passing design, when a new message is created (presumably to later be passed to another process), a special constructor method, called `Create`, is used rather than the normal *C++* `new` operator. The `Create` method is functionally equivalent to `new`, excepting that the memory is allocated from a pre–existing shared memory we call the *shared heap*, rather than from the normal memory heap used by `new`. Each shared heap is made up of a group of heap

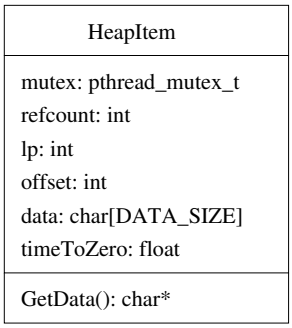

Figure 2: UML model of HeapItem

items. It is in these heap items that the data to be shared is stored. The heap items are indexed according to the offset from the start of the heap. A UML diagram of a heap item is shown in Figure 2. The `Create` method returns a special templated pointer object called a `BPtr` which refers to an individual heap item.

The heap item is made up of multiple items. It stores the number for the logical process it belongs to as well as its index number in the shared heap. The data field is where the actual data for the heap item is stored. The reference count specifies the number of smart pointers that are currently referencing the data in the heap item. This is used to determine when the heap item is no longer in use and can be recycled. This number is atomically incremented and atomically decremented whenever the heap item is copied or when it goes out of scope. Finally each heap item contains a *time to zero* timestamp.

The *time to zero* timestamp is used for simulators which use an optimistic time synchronization algorithm. As discussed previously, there are situations when optimistic simulators need to roll-back due to the generation of a causality error. During this rollback, it may be necessary for the simulator to reacquire dynamically created objects that had previously been freed. This is true regardless of if the simulator is reverting to a previous saved state or performing reverse computation. In order to assist in this process, a heap item will not be immediately available for reuse once its reference count goes to zero. Instead the heap item will store the current simulation time for its LP in the *time to zero* field and will go into a dormant state, preserving itself and the data it contains. The heap item will stay in this state until the *GVT* of the system is greater than its stored *time to zero* timestamp. Only after this happens will the heap item be made available for reuse. Because of this, it is necessary for the simulation to provide the shared heaps updated values for the *GVT* whenever they are calculated.

For simulators that use a conservative time synchronization algorithm, the *time to zero* timestamp is not used. It could either be removed or the GVT for all the heaps in the system could be set to infinity at startup. Either way, heap items that have their reference count go to zero will immediately be available for reuse. Since the concept of rollback does not exist in conservative time synchronization, there is no need to postpone garbage collection.

In our ZeroCopy implementation, the heap items are preallocated. The sizes for the heap items and the number to create can be configured by the user prior to runtime. For example, the user could setup heap items for small blocks (50 bytes), medium blocks (500 bytes), and large blocks (5,000

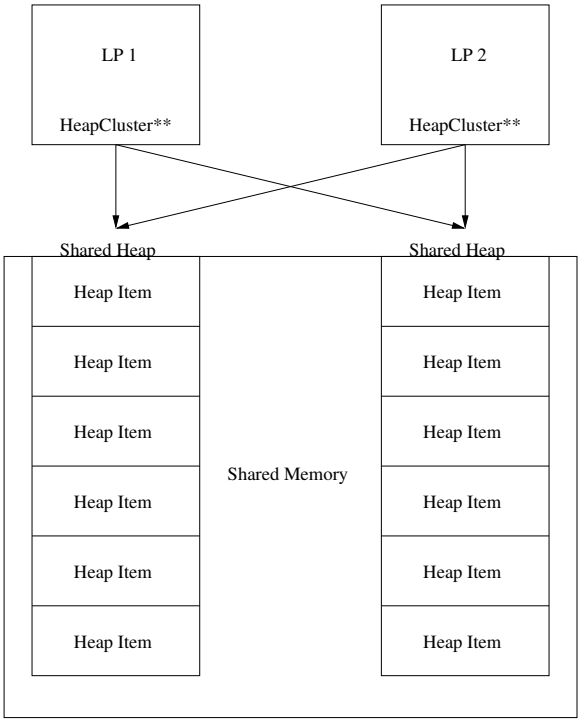

Figure 3: Layout of Shared Heaps

bytes). When an object is created, the heap will return the smallest size that the instance of the object will fit into. By preallocating the blocks, we can provide extremely efficient allocation and deallocation routines since the blocks are of a fixed size and are stored sequentially in memory. When a shared memory pool is requested from the linux kernel, the exact size must be specified and once it is allocated, it cannot be modified. Therefore we feel it makes sense to split the pool into blocks immediately especially since the user of the simulator should have a general idea of the sizes of objects he/she needs to create prior to runtime.

Every logical process in the simulation has its own shared heap, which is initialized at initialization time. The heaps are created in shared memory and the permissions are set so every LP can access every other LP's shared heap as well as its own.

Once all of the logical processes finish creating their own shared heaps, all logical processes make attachments to all other shared heaps. Each LP stores pointers to each of these heaps in a global variable called `HeapCluster`. A diagram of this is shown in Figure 3. Here two LPs have finished setting up their individual shared heaps and have stored pointers to both shared heaps in their *HeapCluster*, which is a global array of pointers to the heap object for all logical processes. In our implementation, each process maintains the array of virtual memory pointers to all shared memory regions in the order of the logical process number of the creating LP. This is shown in figure 4.

It should be noted that while both LPs have pointers to the same shared heap, the actual virtual address of the shared heap will typically be different. When a process attaches to a shared memory segment, the shared memory is mapped into the process's virtual address space and there is no

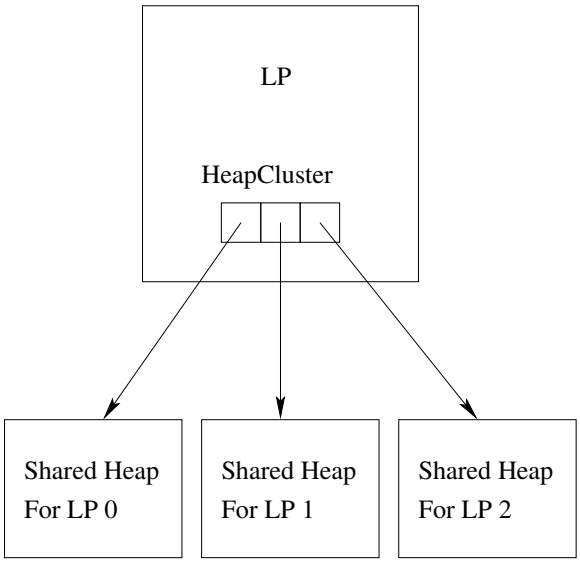

Figure 4: Ordering of pointers in HeapCluster

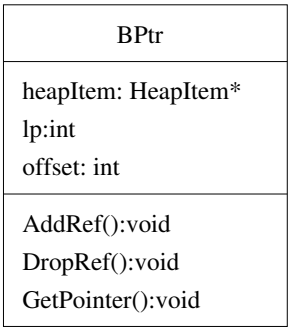

Figure 5: UML model of BPtr

guarantee that it will map to the same location for each process that attaches to it. This is why we cannot simply pass normal C/C++ pointers to data items in the shared memory across logical process address spaces.

Smart Pointers

Similar to the Boost implementation, to reference heap items, our Zero-Copy message passing technique utilizes smart pointers. These smart pointers are necessary because,as discussed previously, standard C type pointers contain a virtual address which may be meaningless when passed between process boundaries. The smart pointers used in the Zero-Copy approach get around this limitation by passing the metadata necessary for the receiving process to obtain the data that it was intended to receive. The smart pointer created for Zero-Copy message passing technique is called `BPtr` and its UML diagram is shown in Figure 5.

The smart pointer is made up of three data items. The integer LP stores the heap number of the

shared heap which contains the actual data being referenced. The offset stores the integer index of the heap item in the data owner's shared heap. The smart pointer also contains a heap item pointer to the heap item referenced by the heap number and offset stored in the smart pointer. This heap item pointer is updated automatically by the smart pointer whenever the smart pointer is copied so it always points to the correct heap item, even when the pointer is passed across process boundaries.

The main difference between our pointer and the Boost offset pointer is that our pointer stores the heap number along with the offset. This offers two major advantages. First it allows our pointers to point to objects in separate shared memory segments. This eliminates many of the concurrency issues experienced with the Boost managed memory segment allocator. The second advantage is that new heaps can be created during runtime if shared memory resources are exhausted. A request is made to the kernel for more shared memory and once it is received the heap structure can be setup and be given a unique heap number. At that point the only thing left to be done is to notify all of the processes to connect to the new heap and add it into their heap cluster.

When a copy of the smart pointer is made, only the heap number and offset data items are copied. Then the `GetPointer` function is called, which gets the heap item pointer. It does this by indexing the HeapCluster array using the LP to acquire a reference to the correct heap and then using the offset to address it to the correct heap item. Another feature of the smart pointer class is that it automatically handles the reference counting for the heap item. The assignment operator and copy constructor have been overloaded to atomically increase the reference count of the appropriate heap item when a new reference to the heap item is made. Similarly, the virtual destructor for any `BPtr` object will decrement the reference count appropriately.

To facilitate the creation of smart pointers, we created a templated `Create` method. This `Create` method first finds a heap item in the heap that is not being used and then uses an in-place `new` to create an instance of the templated class in the heap location. The smart pointer then stores the LP number of the current process and also the offset to the heap item used. Finally the smart pointer increments the reference count for the heap item that it is referencing. The `Create` method can also be overloaded with any argument of any type, which is used to determine which constructor for the underlying data item is to be called to initialize the new object[1].

Once created, the smart pointer behaves syntactically the same as any other pointer. When the smart pointer is dereferenced, it uses the stored LP to connect to the shared heap of the process that created the data and then uses the stored offset to obtain the heap item. The appropriate operators have been overloaded in the smart pointer definition to allow access to all class member functions and data in the referenced class.

Listing 1 above shows an example of creating a Packet object using the smart pointer class. Figure 6 shows a message being passed using a smart pointer. Note that the actual object referenced by the pointer is neither moved nor copied.

[1]We are grateful to the ns–3 development team for the development of their `Ptr` object, which is the basis for much of our `BPtr` implementation

Listing 1: Smart Pointer example

```
1
2  //Create a new packet object using the smart pointer
3  BPtr<Packet> packet = Create<Packet>();
4
5  //The pointer now functions syntactically the same as a regular pointer
6  packet->SetDestIP("192.168.1.5");
7  packet->SetSize("200kB");
8  cout << "My destination IP is: " << packet->GetDestIP() << endl;
9  cout << "My size is: " << packet->GetSize() << endl;
10
11 //The smart pointer class automatically handles reference counting.
12 //Here the reference count is increased to 2.
13 BPtr<Packet> anotherRef = packet;
14
15 //And now decreased back to 1
16 anotherRef = NULL;
17
18 //Below the reference count is decremented, which results in a zero refcount.
19 //The current simulator time is saved into the HeapItem's time To Zero field.
20 // If this isn't done explicitly, it will occur automatically when the smart pointer goes
       out of scope.
21 packet = NULL;
```

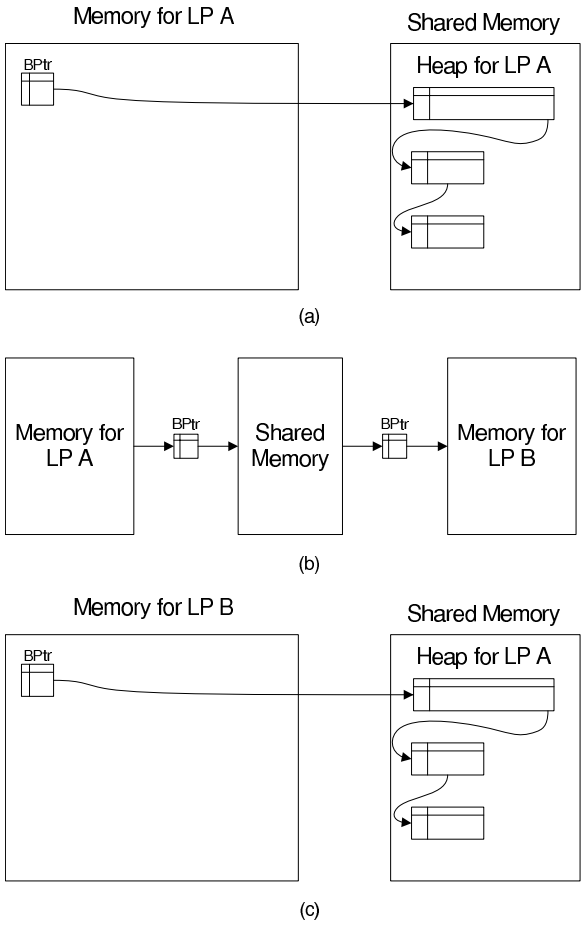

Figure 6: The transfer of a complex object between logical processes using Zero-Copy. (a) The object is created in LP A's shared heap and is accessed by LP A via the smart pointer. (b) The smart pointer is copied to shared memory and then to copied into the memory space of LP B. (c) Using the smart pointer, LB B can directly access the object in LP A's shared heap. The object itself is never moved or copied.

Experiments and Results

SimpleSim

SimpleSim is a very simple distributed discrete event simulator. The simulator enforces causality using a conservative lower bound time stamp (LBTS) algorithm that exchanges timestamp and message count information between LPs in a common shared memory region. The LBTS was calculated using a software rendition of the *Global Synchronization Unit* [4]. Each LP starts off with one event inserted into its event queue. The timestamp for the first event is chosen randomly within the first five simulation seconds.

When handling an event, SimpleSim always creates one new message and sends it to an LP chosen randomly from a uniform distribution. It then examines the size of it's event queue. If the size is less than a predefined constant value, it creates another message which is also sent to a randomly chosen LP. It is possible that the LP can choose itself. Since the size of the event lists for the LPs will grow at approximately the same rate due to the random nature in which the recipients are selected, this prevents unbounded growth in the total number of events for any individual LP.

Included in the message sent to the recipient is a timestamp, a unique ID, and a pointer to an arbitrarily sized chunk of data, which represents the data to be passed to the receiving LP. The timestamps for the new events are chosen from a uniform distribution of 1 to 5 seconds in the future. Added to the timestamp is a predefined constant lookahead value. The unique ID was used mainly for testing purposes.

When SimpleSim receives a new message, it removes the message from the queue and schedules a new event in its event list for the timestamp specified in the message. The individual simulators continue to process and create events until a predefined stop time is reached.

Three versions of SimpleSim were created. The first version is an MPI implementation that copies the entire contents of the message to the receiving process. The other two versions both use a Zero-Copy technique. The second version uses the Boost Interprocess library and the third version uses our Zero-Copy implementation.

We created this simulation because it allowed us to easily vary the size of data being passed between processes. In most cases, the message being passed to the receiving process is a complex object with multiple nested pointers. However in this simulation, the message that is being transferred is uninitialized memory. Thus, there is no serialization step prior to the memory being copied. This also means that there is no de-serialization step. Thus any speedup observed is due only to the lack of a bulk memory copy.

For the first experiment with SimpleSim, the number of LPs was held constant at eight and the size of the message was scaled from 500 to 50,000 bytes. The simulator was set to run for 25,000 simulator seconds. The maximum event list size where LPs stopped sending a second message was set to 5,000 and the lookahead value was set to 5 seconds. All simulation configurations were run ten times and the average result was recorded. Figure 7 shows the run time of the three approaches for a variety of message sizes. As expected, for the two Zero-Copy approaches, the execution time is nearly constant regardless of data size. Again this is because the data is not being copied along with the message but instead is being referenced directly from the shared heap where it was cre-

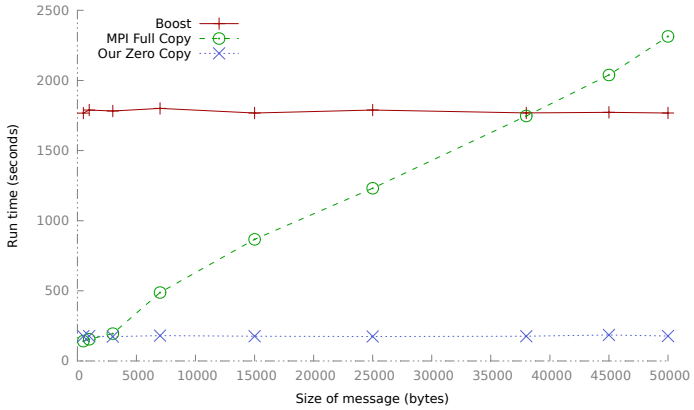

Figure 7: The runtime of SimpleSim with 8 logical processes, varying message size using MPI, Boost Interprocess and our custom Zero-Copy approach.

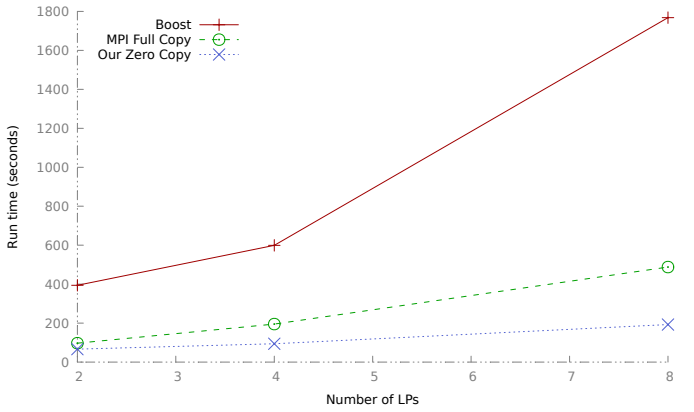

Figure 8: The runtime of SimpleSim with a message size of 7000 bytes, varying the number of logical processes using MPI, Boost Interprocess and our custom Zero-Copy approach. Data collected for 2, 4 and 8 LPs.

ated. By comparison, the execution time of the MPI full copy approach is growing approximately linearly with the size of the data being sent. The results also show that our approach outperforms MPI full copy when the message is larger than approximately 3,000 bytes and outperforms Boost's implementation by almost ten times.

For the second experiment with SimpleSim, we wanted to examine how each simulator instance scaled as the number of LPs participating in the experiment was increased. The size of the data being transferred was fixed to 7,000 bytes. Again the simulator was set to run for 25,000 seconds of simulator time and the maximum event list and lookahead values were set at 5,000 and 5 respectively. Figure 8 show the results of our experiment. The data clearly shows that the Boost implementation scaled much more poorly than either of the other two. This is presumedly because, as discussed previously, the Boost implementation is limited to only one shared memory segment. However even when the Boost version is performing at its peak, our implementation outperforms it by almost six times.

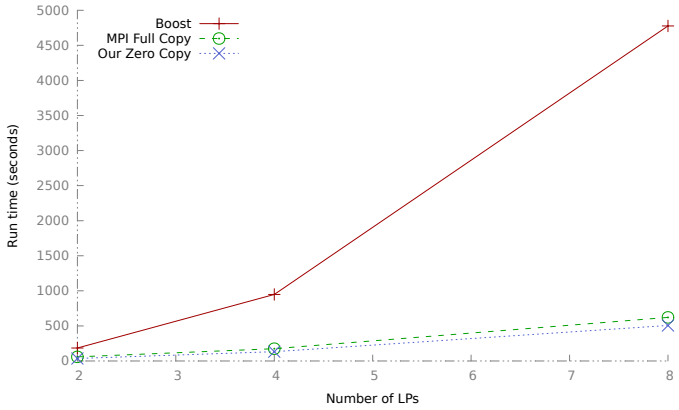

Figure 9: The runtime of GTNetS varying the number of logical processes using MPI, Boost Interprocess and our custom Zero-Copy approach. Data collected for 2, 4 and 8 LPs.

GTNetS

The *Georgia Tech Network Simulator (GTNetS)* is a full-featured network simulator for modeling large-scale topologies. GTNetS offers packet level tracing and models packets with protocol data units (PDUs) that are added and removed as the packet moves up and down the protocol stack. Similar to SimpleSim, GTNetS also uses conservative time synchronization. We chose to test our Zero-Copy approach on GTNetS because the messages passed between processes in GTNetS are complex packet objects that contain multiple PDU objects that must be serialized prior to transfer.

For our GTNetS experiments we created a star topology for each LP. The hubs for each of the stars was then connected to form a clique. Each star was given N-1 nodes where N was the number of LPs participating in the simulation. Each node of a star was configured to send UDP traffic to a node in a different LP. Therefore each LP was sending UDP traffic to every other LP using one of its nodes. Each node was also given a UDP sink to receive data being sent to it. Each UDP packet sent was configured to hold 1,024 bytes of data and each sender was configured with an On/Off Application to use approximately 20% of the available bandwidth. Senders were configured to start at a random time with the first half second of simulation and the simulation was configured to run for 5,000 simulation seconds. All simulation configurations were run ten times and the average value was recorded.

Figure 9 shows the results of this experiment. Again, the Boost version of the simulator scaled worse than the other two versions. Our version of Zero-Copy outperforms the MPI full copy version even though the message size is less than 1,100 bytes. This is due to the fact that the MPI version has to serialize/deserialize the complex packet hierarchy before and after transferring it. This demonstrates that the effectiveness of our approach improves as the complexity of the objects being transferred increase.

Summary

We have presented overviews of two significant prior works, specifically the design and evaluation of the *Global Synchronization Unit* and the *Hardware-supported message passing* approach that demonstrate the ability of closely coupled multi-core sysems to improve overall performance of parallel discrete event simulations. This prior work led us to research a software–only design for zero–copy message passing between separate processes cooperating to execute a distributed discrete event simulation.

Our experimental results show that our new approach to zero-copy message passing can reduce considerably the overall execution time of a distributed simulation when the size and number of inter–process events is significant. The software-only approach can be implemented immediately on any multi–process distributed simulation in a tighly-coupled multiprocessor (multi-core) computing environment.

References

[1] C. Carothers, D. Bauer, and A. Holder, "Scalable time warp on blue gene supercomputers," in *23rd Workshop on Principles of Advanced and Distributed Simulation*, June 2009.

[2] E. W. Lynch and G. F. Riley, "Hardware supported time synchronization in multi–core architectures," in *23rd Workshop on Principles of Advanced and Distributed Simulation*, June 2009.

[3] "Boost c++ libraries @ONLINE."

[4] E. W. Lynch and G. F. Riley, "Hardware supported time synchronization in multi-core architectures," in *Proceedings of the 2009 ACM/IEEE/SCS 23rd Workshop on Principles of Advanced and Distributed Simulation*, PADS '09, (Washington, DC, USA), pp. 88–94, IEEE Computer Society, 2009.