6-1-1999

Parallel DaSSF Discrete-Event Simulation without Shared Memory

James D. Chalfant
Dartmouth College

Dartmouth College Computer Science Technical Report PCS-TR99-346

Parallel DaSSF Discrete-Event Simulation without Shared Memory

James Chalfant
Advisor: David Nicol

Abstract

The Dartmouth implementation of the Scalable Simulation Framework (DaSSF) is a discrete-event simulator used primarily in the simulation of networks. It achieves high performance through parallel processing. DaSSF 1.22 requires shared memory between all processors in order to operate. This limits the number of processors available and the hardware platforms that can exploit parallelism. We are interested in extending parallel DaSSF operation to architectures without shared memory. We explore the requirements of this by implementing parallel DaSSF using MPI as the sole form of interaction between processors. The approaches used to achieve this can be abstracted and applied to the current version of DaSSF. This would allow parallel simulation using shared memory by processors within a single machine, and also at a higher level between separate machines using distributed memory.

1 Introduction

The Scalable Simulation Framework (SSF) is an API that presents a unified interface to discrete-event simulators. Dartmouth has created an implementation of SSF called DaSSF. At present, DaSSF runs on a variety of platforms such as Linux, Irix, and Solaris, among others. The SSF API in its current form does not have all of the functionality necessary to allow for parallel simulation in a distributed memory environment.

1.1 Shared vs. Distributed Memory

DaSSF supports applying multiple processors to a simulation. However, DaSSF 1.22 requires that all of the processors share the same physical memory. This puts certain limitations on when and to what degree parallel processing can occur. First, only hardware that has multiple processors which share memory can exploit parallelism. At present, this is limited to multi-processor Irix computers such as the Origin 2000. Second, the number of processors available is limited to those present within the machine. While it is possible to build (or buy) a machine with more processors, it is generally easier to simply network two existing machines together.

Distributed memory simulation provides a solution to these problems. A distributed memory DaSSF could operate on a number of different computers linked together over a fast network. In this way, single processor machines could still use parallel processing to speed up simulations. Similarly, on architectures where DaSSF already supports parallel processing, if a greater degree of parallelism were desired, distributed memory functionality would provide a straight-forward way to throw more processors at a problem.

1.2 DaSSF-MPI

In order to examine the requirements of parallel DaSSF without shared memory, we implemented a version of DaSSF where inter-processor communication took place solely through message passing. This eliminated all of the shared memory dependencies. The Message Passing Interface (MPI) was used as the communications API. This version of DaSSF (DaSSF-MPI) was primarily developed on a network of Linux machines connected using Myrinet. It has also been compiled to run on the multi-processor Irix machines that DaSSF 1.22 already supports.

DaSSF-MPI fulfills both short-term and long-term goals. As an end unto itself, DaSSF-MPI reduces the hardware requirements for parallel DaSSF simulations. Although the performance of DaSSF 1.22 on the Origin 2000 is superior to that of DaSSF-MPI on a Myrinet network with a similar number or processors, DaSSF-MPI still works at a respectable pace and a fraction of the hardware cost. Furthermore, adding extra processors is as simple as attaching more machines to the Myrinet switch. Compiling DaSSF-MPI on the Origin 2000 also allows us to examine the overhead of MPI vs. shared memory on this platform.

Looking farther ahead, adding distributed memory functionality to DaSSF without eliminating the possibility of shared-memory parallelism would represent the best of both worlds. DaSSF-MPI represents a step towards this goal. A model could be distributed at one level among processors within a single shared memory machine, and then again at a higher level across a network of such machines. This could be accomplished by using TCP/IP instead of MPI (or perhaps an MPI implementation that uses TCP/IP), or more probably by a higher level simulation communications package such as the High Level Architecture Run-Time Infrastructure (HLA RTI). This proposed version of DaSSF could apply a whole network of Origin 2000s to a single simulation.

1.3 Thesis Outline

In Section 2, we provide a brief introduction to SSF. Section 3 extends this to an overview of DaSSF from the modeler's perspective. DaSSF from the simulator programmer's view point can be found in Section 4. This covers the software structure that underlies the simulator. Section 5 illustrates where and how DaSSF 1.22 is shared-memory dependent. The means DaSSF-MPI uses to operate correctly in a distributed memory environment are covered in Section 6. How this work might be applied to future versions of DaSSF is covered in Section 7. Results of running identical models on DaSSF 1.22 and DaSSF-MPI on different platforms are compared in Section 8. Future work is outlined in Section 9. Final conclusions is discussed in Section 10.

1.4 Acknowledgments

DaSSF was created by Professor David Nicol and Jason Liu. This thesis is derived wholely from their previous work.

1.5 Related Work

Related work in parallel and discrete-event simulation has been done by a number of others in a variety of different settings. Work in parallel discrete-event simulation has been done by R.M. Fujimoto [3]. P. Heidelberger and D. Nicol[4][5] [6] have used parallel discrete-event simulation when simulating Markov chains. D. Nicol and S. Roy[7] have simulated petri nets with a parallel discrete-event simulator. Languages for designing parallel discrete-event simulator models have been developed by R.L. Bagrodia and W.T. Liao[1] and B.R. Preiss[8]. D.O. Rich and R.E.

Michelson[2] have analyzed the shortcomings of efforts to apply standard simulation modeling tools to parallel simulations. J.S. Steinman[11] has presented an environment for parallel discrete-event simulation.

2 SSF

The Scalable Simulation Framework is an API created by the S3 Consortium as a means of providing a generalized interface to discrete event simulators. SSF is designed to allow portability of models across SSF-compliant simulators. In addition, it provides a separation between model builders and simulator writers.

2.1 SSF at a Glance

The SSF framework consists of a number of base classes. A simulation is comprised of interactions between these classes and others derived from them. The SSF base classes are class **Entity**, class **process**, class **inChannel**, class **outChannel**, and class **Event**.

The basic unit of a model is an **Entity**. Communication between Entities takes place through the transfer of **Events**. An Event must travel through a channel from one Entity to another. A channel begins at one Entity's **outChannel**, and terminates at another Entity's **inChannel**. All of an Entity's actions, including those based on incoming events, are handled by its **processes**.

It is somewhat easier to see these relationships in Figures 1 and 2, which represent a 2-input multiplexor. Here, the multiplexor and the traffic generators are **Entities**. Where the channels leave the traffic generators and the multiplexor, **outChannels** exist. The two channels from the traffic generators terminate at the multiplexor in **inChannels**. Inside each traffic generator, there is a **process** which randomly generates traffic to be sent to the multiplexor. Another **process** exists in the multiplexor, which waits for traffic from the traffic generators, and then forwards that traffic onto the single exiting channel when it arrives. The traffic itself is made up of **Events**. When the **process** in the traffic generator determines it's time to send some traffic, in makes a number of **Events** and writes them to the **outChannel**, which is the starting point of the channel leaving the traffic generator.

2.2 SSF Entity

An **Entity** is an object, such as a node in a simulated grid of processors or a switch in a multi-layer multiplexor. It generally has a number of inChannels, outChannels, and processes, which it *owns*. An Entity may also *contain* other Entities. This ability normally leads to a tree of entities which contain one another, with a root Entity at the base.

2.3 SSF process

A **process** is the means by which an Entity acts. The body of a process is code fragment that represents the actions of a simulated Entity. The process's body repeats itself, so that once the logic it represents finishes, it starts again from the beginning. That logic may be as simple as creating an Event, writing it to an outChannel, and pausing for 5 units of logical time. If this were a process body, that process would create and write an Event every 5 time units until the simulation halted. A process can interact with the simulation run-time system in a variety of ways. It can wait until a user-defined function returns true, using the `waitUntil()` method. It can pause

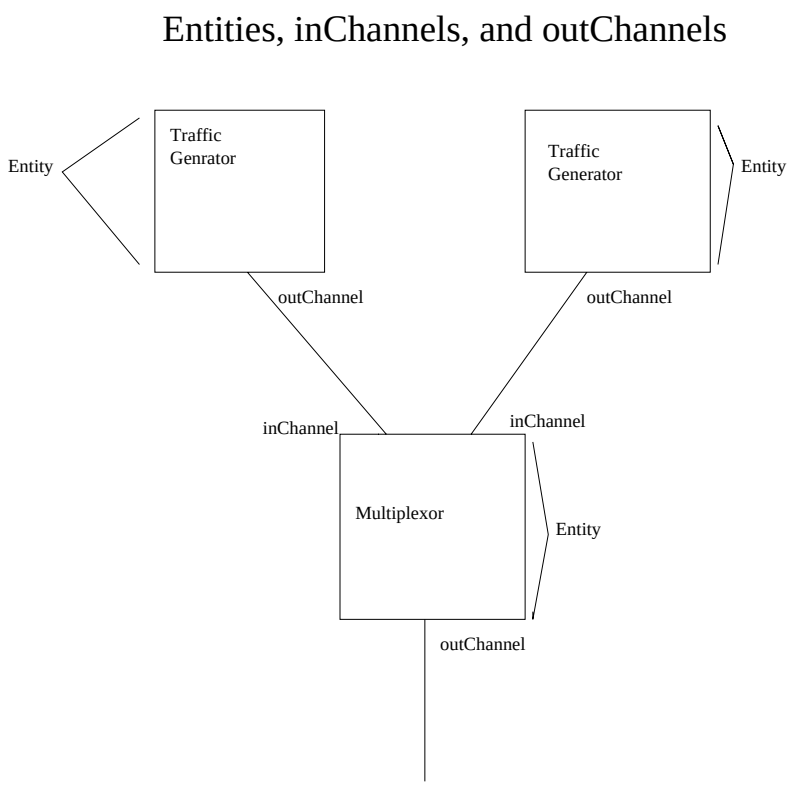

Figure 1: Here we we see the Entities, inChannels, and outChannels present in the above example

for a specific amount of logical time with the `waitFor()` method, or until the end of the simulation, using `waitForever()`. More interestingly, it can wait for the next Event to arrive on an inChannel or set of inChannels, using `waitOn()`. The `waitOnFor()` method functions like `waitOn()`, but stops waiting if a set amount of time passes. Finally, a process will implicitly wait if it calls logic that requires the simulation framework to suspend it. For instance, this occurs while writing an Event to an outChannel. It should be noted that logical time does not necessarily advance between every process suspension and activation. A process can suspend and later awaken at the same logical time. Consider a process which writes an Event to an outChannel, causing the process to suspend at time T. The simulation framework then takes over, handling the underlying work that is necessary when an Event is written to an outChannel, such as placing the Event in a queue of items to be delivered to the appropriate inChannel. When this is done, the suspended process is awakened and continues operating. The logical time has not advanced; it is still T.

2.4 SSF inChannel, outChannel

inChannels and **outChannels** are rather straightforward. An Entity may own a set of each. Events are written to outChannels and arrive on inChannels. An outChannel is mapped to a specific inChannel using `mapto()`. It is possible to map multiple outChannels to one inChannel. Likewise, an outChannel can be mapped to any number of inChannels. An outChannel can be mapped to an inChannel even if both are owned by the same Entity. By using the outChannel

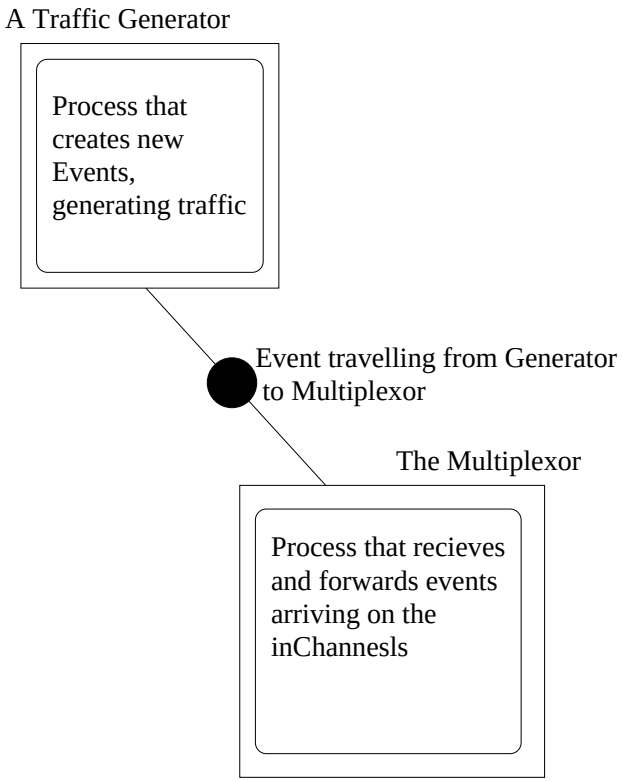

Figure 2: In this expanded view, we see the Processes that represent and Entity's actions and an Event that has been generated by them.

method `minDelay()`, a delay can be associated with an outChannel. This means that any Event written to that outChannel will be delayed a set amount of logical time before it arrives at the appropriate inChannel. Events are written to an outChannel using the `write()` method. It is also possible to introduce an additional delay when an Event is written to an outChannel. This delay is specific to the event, however, not the outChannel.

2.5 SSF Event

The basic unit of inter-Entity communication is the `Event`. An Event is extremely simple, consisting of only a send time and a receive time. These can be viewed through the `sendTime()` and `recieveTime()` methods.

2.6 Timelines, Alignment, and Containment

An interesting aspect of SSF relates to the concepts of timelines and alignment. A **timeline** is essentially a discrete-event simulator. Everything a timeline is responsible for simulating has a logical time equal to that timeline's current time. When an Entity's logical time is determined by a given timeline, it is said to be *aligned* to that timeline. Alignment of an Entity to a timeline is accomplished with the `alignTo()` method. After it is called, the logical time of that Entity and everything it owns is then determined by that timeline. It is important to note the distinction

between *ownership* and *containment*. Ownership refers to an Entity's processes, inChannels, and outChannels. The logical time of these is determined by an Entity's alignment. Containment refers to an Entity's children, which are other Entities. Alignment of the parent Entity specifies nothing about them. Therefore, if a timeline is at logical time T, every Entity aligned to that timeline and all the processes, inChannels, and outChannels owned by these Entities are also at time T.

The difference between ownership and containment is significant because SSF allows for the existence of multiple independent timelines. That is, timelines `X` and `Y` can both exist within a given simulation. Timeline `X` can be at logical time `T1` while timeline `Y` is at time `T2`. `T1` and `T2` are completely independent, so `T1` can be earlier or later than `T2`. Furthermore, this may change during the course of simulation. Some constraints exist, though, when Entities on these timelines wish to interact. Consider Entities `A` and `B`, with `A` aligned to `X` and `B` aligned to `Y`. If an outChannel from `A` is mapped to an inChannel on `B` and a minimum delay `D` is associated with that outChannel, then the simulation framework insures that timeline `Y` is never more that `D` units of logical time ahead of timeline `X`. In this way, Entities can be independently simulated within certain limits. SSF allows for the existence of outChannels with a delay of zero. However, if a zero-delay channel crosses between two timelines, this effectively makes them one timeline, since one can never be more than zero units of logical time ahead of the other.

Entities become aligned to timelines in a variety of ways. The first, as mentioned above, is with the `alignTo()` method, where one Entity becomes aligned to another. By using `makeIndpendent()`, a new, independent timeline is created and the Entity is aligned to it. Finally, if no alignment at all is specified, an Entity is aligned by default to its parent. If no parent exists, a call to `makeIndpendent()` is made. It should be noted that this is the only function of containment: to provide a default alignment scheme.

2.7 Model Construction in a Flash

Model construction takes place via an initialization loop. Both Entities and processes have `init()` methods. Processes, child Entities, outChannels and inChannels can all be created within an init() method, as well as mappings between channels. In terms of the order in which init() calls are made, the only guarantee is that an Entity will be `init()`ed before any process it owns.

3 DaSSF

The Dartmouth Implementation of SSF provides both more and less functionality than that outlined is the SSF specification. Some features, such as dynamic entity realignment, are unsupported in DaSSF 1.22. DaSSF does provide a number of extensions to the specification though, such as multiChannels, random number generation, and semaphores. Finally, DaSSF makes parallel SSF simulations possible.

3.1 Departures from the SSF specification

DaSSF departs from SSF in a number of ways. The process class becomes the Process class, discarding the puzzling lowercase name present in pure SSF. A number of Entity methods used in starting the simulation in SSF, such as startAll() and joinAll(), aren't used at all (this is why they weren't covered in Section 1). Instead, DaSSF constructs a model in a custom fashion. Finally, all containment, alignment, and channel mapping must be established before the simulation begins. This is a fairly serious departure from pure SSF, where all of the above can change during simulation.

3.2 Constructing a model in DaSSF

Constructing a model in DaSSF is rather different than in SSF. The base of a DaSSF model is the `make_root()` method. `make_root()` is user defined. It returns an object derived from class **Entity_Root**, which is DaSSF-specific. In a DaSSF model, one Entity_Root exists per processor. An Entity_Root differs from a normal Entity in that it goes through a third phase after construction and initialization before simulation begins. This is the *build* phase. DaSSF supports parallel processing, but this complicates model construction. For instance, if we wish to map Entity `A`'s outChannel to an inChannel on on Entity `B`, but `A` is on processor `P1` and `B` is on processor `P2`, how does A know if `B` has even been constructed yet? DaSSF's answer is to create all Entities in the construction and initialization phases, and postpone all channel mapping until the Entity_Root's build phase. Barrier synchronization takes place between the framework's calls to `init()` and `build()`,so every Entity_Root has finished its initialization before any enters the build phase.

The `make_root()` method is given argc, argv, the id of its processor, the total number of processors, and the simulation end time as arguments. This allows the modeler to explicitly partition the model onto different processors, based on their availability. In order to access shared memory, DaSSF uses a global naming scheme. All SSF objects are created within a shared area of memory. The method `addGlobalName()` allows a modeler to associate a memory location with a character string. The `resolveGlobalNames()` method causes all processors to have an identical list of global names, as well as the memory addresses associated with each name. Naturally, no new global names can be added after this is called. To retrieve an address using a global name, the method `getGlobalAddress()` is used. In this way it is possible to construct a model across different processors.

3.3 Processes and Augmenting the Model's Source Code

DaSSF handles the threading of Processes internally. A Process is suspended and restored using an internal heap, rather than relying on the operating system for these tasks. This requires augmenting the model's source code with extra C++ code before compiling it, which is accomplished by processing the source with a perl script. Some limitations on the allowable C++ syntax result, but these restrictions are rather small. Class declarations that are derived from SSF base classes, functions that are going to be used as the bodies of SSF Processes, and state variables within these functions all must be marked by the modeler. Usually this is done by adding an appropriate comment before them, such as `//!SSF PROCEDURE`.

3.4 Memory Management and Events

Events have reference counters to determine when they should and shouldn't be deleted. Therefore, explicit deletes aren't used with Events. Instead the `unreference()` method is called. When an event is passed to the SSF framework using a write() call, it becomes the framework's responsibility to manage the Event's memory. Simply put, explicitly deleting or unreferencing an event after writing it to an outChannel would be bad. Once an event is delivered back to the model by arriving on an inChannel, it is once again the model's responsibility, and here is where `unreference()` should be called.

3.5 DaSSF Extensions

DaSSF provides a number of extensions to SSF. Several of these are rather straight-forward, such as provisions for robust random number generation, semaphores, and timers. These are made available with through class **Random**, **Semaphore**, and **Timer**, respectively.

A number of convenience extensions are also present. It is possible to use an Entity's method as the body of a Process, so that the Entity's class variables can be accessed without using pointers. The `waitsOn()` and `waitsOnFor()` methods create a static list of inChannels for a process to wait on. This means that a process can call `waitOn()` without specifying any inChannels, and the process will automatically wait on the static list. A `waitUntil()` method has also been added that takes an absolute time as an argument. This allows Processes to wait until a given time rather than for a specified interval.

The class **multiChannel** is probably the most interesting SSF extension. It is derived from the outChannel class. While it is possible to map a single outChannel to multiple inChannels, the same delay will exist along each mapping. A multiChannel allows the modeler to associate a different delay with each inChannel the multiChannel is mapped to. The benefits of a multiChannel over a number of outChannels lies in space efficiency and ease of model design. With a multiChannel, only one Event exists per timeline, and this Event is passed from inChannel endpoint to inChannel endpoint on that timeline, based on the delays associated with each. This reduces the memory used.

3.6 DaSSF Runtime Options

DaSSF contains a number of options that can be set at runtime. These are -debug, -debug_value, -nprocs, -endtime, -heap, -serial, -seed, -steal, -stealevel, -automap, -backstore, and -stats. -debug and -debug_value can be used to create debugging output. The number of processors to use for parallel simulation is set with -nprocs, and -endtime sets when the simulation will stop. The per processor heap size (in megabytes) is set with -heap. -serial constrains the simulation to serial (as opposed to parallel) simulation. The -seed option sets the random number seed. A file specified by -backstore will be used as a space for shared memory when multiple processors are used. It is Irix specific. -steal and -stealevel are used to turn on and set the threshold of stealing, which is a form of dynamic load-balancing. Finally, the -automap option lets a model be partitioned among processors by the system instead of the modeler, by mapping new timelines to processors in a cyclic fashion.

4 DaSSF: How it works

We will now discuss how DaSSF functions "under the hood", below the API which the modeller sees. This is necessary to understand which areas of DaSSF are dependent on shared memory to work properly. It also is useful in appreciating the changes we made to remove these dependencies and implement DaSSF using MPI.

At its heart, DaSSF is just a normal discrete-event simulator. Simulations tend to advance time in one of two ways: by using discrete events or via a time-step. A time-stepped simulator with a timestep of size X advances from time T to T+X, updates everything that is being simulated, and advances time by X again. A discrete-event simulator keeps track of events that take place in the future. It moves to the next event scheduled to happen, simulates that event happening, and then advances time to the next scheduled event.

A discrete-event simulator has a structure containing future events to be processed, ordered by the time when the events will occur. We'll call this time the event's timestamp. The structure holding the events we'll refer to as an event-queue, although it isn't necessarily a queue (it could just as easily be a heap, for instance). The simulator looks at the event in the event-queue with the lowest timestamp. It advances the simulation's logical time to that timestamp and removes that event from event-queue. Then, the event is processed, which often triggers future events. By giving these future events a timestamp that equals when they should occur and placing them in the event-queue, the events become scheduled to happen. After it has been processed, the original event is discarded. The simulator then advances logical time to the next event's timestamp, removes it, and repeats. Obviously, events with timestamps earlier than the simulation's current logical time are illegal.

4.1 DaSSF in Action

DaSSF is somewhat more complicated than a basic discrete-event simulator. We will first examine how DaSSF functions when a simulation is running. This requires that an SSF model has already been constructed and is in place. We will explain how this is accomplished later.

4.1.1 SSF through discrete-event simulation

A single timeline can be treated as a normal discrete-event simulator. For purposes of discussing a discrete-event simulator running an SSF simulation, we'll refer to simulator events as events and instances of the SSF class **Event** as SSF Events. Entities on a timeline send SSF Events to other Entities, run Processes, and generally do things allowed by the SSF specification. Each of these actions can be modeled by a discrete event.

Let's consider an example of this: If a Process P wishes to wait for 10 units of logical time, it can place a "wake me up" event in the event-queue. When the simulator reaches that event, it wakes up P and runs it until it suspends again, perhaps by waiting on inChannel I. Later, when some other Process is woken up, it writes an SSF Event to an outChannel mapped to I. This creates an event that is analogous to an SSF Event. When the simulator processes this event, it notices that the P is waiting on that I, so in addition to delivering the SSF Event it wakes up P and runs it until it suspends once more.

Clearly, this isn't sufficient to handle a full SSF simulation. What happens when an Entity wishes to send an SSF Event to something that isn't aligned to the same timeline? Here the concept of a simulation window comes into play.

4.1.2 Simulation Windows and multiple Timelines

A simulation window is essentially the length of time a timeline can perform discrete-event simulation without worrying about what other timelines are doing. Suppose timeline X know that nothing happening on timeline Y can effect it until time T. X can then safely process all the events in its event-queue that have timestamps earlier than T.

We established earlier that when Entities on different timelines have channels between them, these channels must have some delay. Consider the timeline-crossing channel with the smallest delay, which we will call MD (for minimum delay). A simulation window exists for all timelines from time 0 until MD. Let's look at this with an example. Entity A is on timeline X, and Entity B in on timeline Y. X and Y are between times 0 and MD. A owns outChannel M, which is mapped to B's inChannel N. X simulates an event that causes A's Process P to write SSF Event E to M.

This should cause an event e to be placed in Y's event-queue at time T+D, where T is X's current time and D is M's associated delay. However, D $\leq$ MD. Therefore, e is scheduled for time T+D, which is certainly later than MD. Thus, Y can ignore events created on other timelines up until time MD. Let's consider e again. Because the event that created e took place between times 0 and MD, e can't take place until MD or after. Similarly, if the event causing e took place after MD, e would have to be scheduled for a time greater than 2*MD. The point is this: all the events that X creates for Y's event-queue that are scheduled for time T $<$ 2*MD have been created before MD. If timelines X and Y pause at MD and exchange all of the events they've created for each other with timestamps $<$ 2*MD, this will be sufficient to insure proper behavior.

This can be generalized to the rest of the simulator. We have an arbitrary number of timelines. The size of the simulation window is MD. Every timeline should pause at each multiple of MD, and wait until the other timelines also reach this point. Therefore, timelines synchronize at times MD, 2*MD, 3*MD, etc. We'll call an event created by one timeline that is meant for another timeline an off-timeline event. In the above example, e is an off-timeline event. When the timelines synchronize at time x*MD, they send all of their off-timeline events with timestamps $<$ (x+1)*MD to their respective destination timelines.

DaSSF actually has two structures for future events. Each timeline has an event-queue that stores events with timestamps less than the end of the simulation window. Events that are created with timestamps larger than this are put in a second structure. This structure makes it simple to access all the events that take place during a given simulation window. Conceptually, though, it isn't any different from the scheme covered above.

This is sufficient if all of the timelines are running in parallel. However, for this to actually occur there would have to be a processor per timeline, which is generally not the case. Therefore, in addition to dealing with off-timeline events, we must also insure that every timeline is attended to when multiple timelines exist on one processor.

4.1.3 Multiple Timelines on one Processor

DaSSF solves this problem by keeping a queue of running Processes. This Process-queue has at most one Process for each timeline on that processor. The bottom-most Process is removed from the queue and run until it suspends. When it suspends, the timeline that this Process is aligned to takes over. It begins to simulate events, and continues until a Process on that timeline becomes scheduled to run, or until logical time reaches the end of this simulation window. If the timeline reaches a point where a Process should start running, this Process is placed at the end of the Process-queue. Control then passes back to DaSSF, which removes the next Process on the bottom of the Process-queue and repeats. In this way, each timeline is given some processing time in a cyclic fashion. This continues until no more Processes are scheduled to execute. When this happens, it means that the logical time on each timeline has reached the end of the simulation window.

That's a general overview of how DaSSF runs. An SSF Model is distributed among multiple processors based on the timelines present. Each timeline is assigned to a processor, which often results in multiple timelines on the same processor. By using the Process-queue, each timeline is advanced until they are all at the end of the simulation window. When every timeline on every processor is at the end of a simulation window, synchronization takes place. Off-timeline events are distributed, both between different timelines on the same processor and between timelines on different processors. Every timeline that has some event scheduled to occur between the current time and the next simulation window places a dummy-Process in the Process-queue, and simulation restarts. The dummy-Process does nothing except guarantee that at some point control will pass

to the timeline it is aligned to.

4.2 Constructing the Model

We've covered how a Model is simulated once everything is in place. How do we reach this point, though? How are all of the Entities created and aligned to timelines, and where are the outChannels mapped to inChannels? Finally, what does the modeler have to do?

These answers can be seen by following the DaSSF startup sequence. We will examine everything that happens up until the loop outlined above takes over and simulation time begins to advance.

4.2.1 Basic Initialization

The first few steps are extremely simple. The command line is parsed. A large chunk of shared memory is allocated, and divided into sections for each processor. A number of read-only global variables are initialized, such as the number of processors present or whether debugging output should be created. Then, a child process is forked from the main process onto each of the participating processors. A number of processor specific variables, such as the Process-queue, are created. All structures are created within shared memory.

4.2.2 make_root(), DaSSF begins to run

Non-trivial things begin to happen with the call to make_root(). As noted earlier, this is where a DaSSF model begins. make_root() returns an instantiation of class Entity_Root(). A new timeline is created, and the root entity is aligned to it. Each processor then manually creates a Process for each root, the body of which is function root_process(). This root Process is placed on the Process-queue.

At this point, the outer loop of DaSSF begins to run. This causes the bottom-most Process on the Process-queue to be removed and executed. This is the root Process. However, a global flag is set so that the root Process's timeline doesn't gain control when the root Process suspends. Instead, the next Process is removed from the Process-queue. At that point, the model will be fully constructed.

4.2.3 Entity Construction in Detail

Before we cover what goes on in the root Process, it is important to understand what exactly happens when an Entity's constructor is called. Along with the modeler-defined constructor, there are also a number of things that happen internally. Most notably, an init_E_event is either created or modified. An init_E_event is an instantiation of the class **KernelEvent**. This is the same class as that used for events in timelines. Here, however, it serves a different function. When a new Entity is constructed, if no init_E_event exists, one is created. A KernelEvent can point to an Entity. In this case, it is set to point to the newly created Entity. In DaSSF, Entities can also point to other Entities, to ease the construction of linked lists of Entities. When an Entity is constructed and an init_E_event is already present, the init_E_event is changed to point to the new Entity, and the Entity then points to what the init_E_event used to reference. In this way, the init_E_event is the beginning of a list of all the Entities that have been constructed on a given processor. When we called make_root() earlier, we called an Entity constructor (the Entity_Root's), so an init_E_event now exists. It possible (though not necessary) that other Entities were created within this constructor, so the init_E_event may represent a number of Entities at this point.

4.2.4 The root Process: init()

Now we proceed into the root Process. The first Entity that the init_E_event points to is removed, and the init_E_event is updated to reference the next Entity in the list. The removed Entity's init() method is called. This could result in other Entities being constructed, which would modify the init_E_event accordingly. When the Entity's init() method finishes, the init() method for every Process currently owned by that Entity is called. In this way, we comply to the SSF specification that an Entity be init()ed before its Processes are. This procedure is repeated until the init_E_event doesn't point to an Entity. In this way, we init() the root Entity, all of the Entities created by the root's constructor and init() method, and every Entity created by their constructors and init() methods. We've also init()ed any Processes that were created before the owning Entity's init() method was called.

4.2.5 The root Process: build()

At this point, no channel mappings should exist. This is because a DaSSF compliant model only maps channels in the build() method of the root Entity, which has yet to be executed. Once a processor finishes the above Entity initialization, it waits until all other processors reach the same point. Each then proceeds to call its root Entity's build() method. In this way, we guarantee that Entities which exist on other processors have been created before we try and map to their inChannels.

4.2.6 The root Process: init() revisited

Once every processor completes the build phase, the Entity initialization sequence takes place again. This is because it is possible to create an Entity within the build phase which needs to be init()ed. However, Entities created earlier can't make assumptions about the existence new ones created here, so mapping to them would be difficult.

Every Process that was created before its owning Entity was init()ed has been init()ed. However, it is possible to create a Process for an Entity after that point. For instance, a Process could have been created in the build() method. For this reason, we enter a Process initialization loop. An init_P_event exists, which is analogous to the init_E_event. If a Process is constructed, but it's owning Entity is already init()ed, then it becomes part of a list of Processes that starts with the init_P_event. The root Process then runs through this list, init()ing each Process in it.

4.2.7 Alignment in the root Process: when and where

At this point, all Entities and Processes that are supposed to exist when the simulation starts have been constructed, init()ed, aligned, and all of the channel mappings have taken place. We haven't explicitly discussed alignment yet. It is the modeler's responsibility to align an Entity after it has been constructed. If an Entity's init() method is called and the Entity is not yet aligned, it is assumed that the modeler does not wish to align it. In this case, it is DaSSF's responsibility. Here containment comes into play. If the Entity is contained by a parent Entity, then it is aligned to the same timeline as the parent. If there is no parent, a new timeline is created and the Entity is aligned to that. Therefore, once the the Entity initialization sequence has been completed alignment has been dealt with, either by the modeler or by DaSSF.

4.2.8 The root Process: starting DaSSF

The model has been built at this point. All that remains is to start it. Once an Entity has been initialized, it gets placed in a list. At this point, this list should contain every Entity that has been created. The root Process then goes through this list, and calls the start_all() method of each Entity. start_all() adds each Process to the Process-queue, and then executes each Process. However, since there is global flag set that prevents timelines from taking control of the simulation, executing these processes does not cause logical time to advance.

Finally, the global flag is unset, and the simulation begins in earnest. Processes are removed from the Process-queue, and timelines begin to advance in logical time.

4.3 Stealing and automapping

Stealing is a feature of DaSSF we haven't been covered yet. Stealing is a form of load balancing. Basically, when a processor finishes advancing all of its timelines to the end of the simulation window, it enquires if the other processors are done. If they aren't, it makes a request to take over some of the processing. It does this by "stealing" the Process at the bottom of the Process-queue on another processor. In this way, the processor takes over the timeline the stolen Process is aligned to. Everything is kept in shared memory, so stealing a Process is as simple as removing it from the Process-queue and telling the "stealing" processor its memory address.

Automapping is when the system takes care of partitioning the model across processors. It is done by assigning each new timeline to a different processor, instead of the processor on which it was created. Again, because memory is shared, this consists of merely telling the assigned processor the timeline's memory address.

4.4 KernelEvents and MultiChannels

Class **KernelEvent** represents the events used in discrete-event simulation by the timelines. They represent all of the actions that can take place in DaSSF. There are 9 types of KernelEvents: Idle, Channel, MultiChannel, Timeout, NewProcess, RunProcess, Canceled, Init_Entity, and Direct.

4.4.1 KernelEvent Types

Idle events are unused in DaSSF. Channel events represent the arrival of an SSF Event on an inChannel. In DaSSF, when a Process writes to an outChannel, one of two things happens. If the delay on the outChannel is small enough that the SSF Events will arrive before the end of the simulation window, a Channel-type KernelEvent is created for each inChannel that outChannel is mapped to. These new KernelEvents are then placed into the event-queue. If, however, the delay is so large that nothing will arrive until a future simulation window, a KernelEvent representing the write is inserted as a single future event. When that simulation window is reached and events are distributed to other timelines, that KernelEvent will be transformed into a number of different Channel-type KernelEvents, each representing an SSF Event arriving on an inChannel. Technically, the KernelEvent representing the write is also a Channel-type one, but it only exists when the SSF Event won't arrive until a future simulation window. MultiChannel KernelEvents are somewhat more complicated than the other types, and will be discussed later. Timeout KernelEvents stop a Process from waiting on an inChannel. These are caused when the time limit on a waitOnFor() expires. NewProcess and RunProcess KernelEvents represent the creation of a new Process or the awakening of an existing Process that was suspended until a specific time. Canceled KernelEvents

are ones DaSSF should ignore and delete, which is occasionally necessary for some internal operations. Init_Entity-type KernelEvents represent an Entity that has been created after the simulation begins. It is like an init_E_Event, except the timeline handles it instead of the root Process. Direct KernelEvents are for a legacy feature no longer supported in DaSSF.

4.4.2 MultiChannels and MultiChannel-type KernelEvents

MultiChannels differ from normal outChannels in that they allow a different delay for each inChannel they map to. Internally, a multiChannel organizes the inChannels it's been mapped to by the timelines those inChannels are aligned to. For each timeline, an instance of class **_TimeLineTarget** exists. This has a list of inChannels on the targeted timeline that this multiChannel maps to, along with the delay associated with each.

A MultiChannel-type KernelEvent differs from normal KernelEvents in that it points to a _TimeLineTarget. Other KernelEvent types point to SSF objects, such as a Process (in a RunProcess KernelEvent) or an inChannel (in a Channel KernelEvent). KernelEvents often point to SSF Events as well. Thus, a Channel-type KernelEvent points to an SSF Event and the inChannel it will arrive upon. A MultiChannel KernelEvent points to a _TimeLineTarget, which is DaSSF specific. This is done because it allows for greater efficiency. When a MultiChannel KernelEvent is present on a timeline, that one event represents all of the SSF Events that will arrive on all the the inChannels in that _TimeLineTarget. This is because once the MultiChannel KernelEvent arrives on the inChannel with the shortest delay, it reenters the event-queue with a new timestamp equal to the time it should arrive on the next inChannel. In this way, only one MultiChannel KernelEvent needs to be present in the event-queue.

5 Dependencies on Shared Memory in DaSSF

DaSSF achieves high performance by operating in parallel. In order to do this, data must be shared between processors. If memory is also shared, this is very easy. Before we can discuss how DaSSF might be altered to run without shared memory, we must cover how and where the current version is dependent upon it.

5.1 Dependencies during Simulation

We will examine DaSSF shared memory dependencies in the same way we covered DaSSF operation. First, we will look at what sections of DaSSF use shared memory while the simulation is running. Model construction will be analyzed after this.

5.1.1 Simulation vs. Synchronization

Model simulation can be divided into two steps: the discrete-event simulation that takes place during simulation windows, and the synchronization that happens at window boundaries. By the definition of a simulation window, information generated by other processors is irrelevant during one. Timelines only become aware of events created elsewhere during the synchronization at window boundaries. Therefore, the simulation that takes place during a simulation window doesn't rely on memory being shared. Processes can be removed from the process-queue and events can be simulated on timelines in a distributed memory environment without change.

The synchronization step that takes place at a simulation window boundary is rather different. Here, KernelEvents are transferred from one processor to another, and timelines can be accessed by any processor in the machine. Obviously, DaSSF relies on shared memory rather heavily here.

5.1.2 Synchronization in Detail: handle_events()

The synchronization step is a procedure called handle_events(). The shared memory dependencies can be seen with a detailed look at how handle_events() works.

The Event-List: Earlier, we noted that two structures for future events exist. One holds events that take place before the end of the simulation window. The other holds events farther in the future; those that lie beyond the window boundary. The first exists for every timeline, and can be considered that timeline's event-queue. The second is a per-processor structure. We'll call this the event-list. The event-list is actually a series of event storage structures. Each sub-structure represents all the events that have been scheduled for a given simulation window. Consider an event-list with sub-structures S1, S2, S3,...,SN. If an event is created with a timestamp that places it in the next simulation window (instead of the current one) it is placed in S1. If it will take place two simulation windows from now, it goes into S2, and so forth. If the timestamp is so large that it overflows the event-list, it is stored elsewhere. When time advances to the next simulation window, S2 becomes S1, S3 becomes S2, etc. Since S1 becomes SN, we don't lose space when this happens. The sub-structures are re-indexed rather than actually moving their contents from one to the next, so this is a rather inexpensive process. When simulation time advances enough, events in the overflow structure are transfered into the event-list.

Determining the Simulation Window: The first step of handle_events() is to find the next sub-structure in the event-list that contains events. Think about the case where S1, S2, and S3 are all empty, and the next event to take place resides within S4. In this case, time should advance all the way to the simulation window S4 represents, instead of wasting time in the intervening ones. This has to be done across all processors, though. Even it processor 1 has no events in the event-list for 3 simulation windows, processor 2 may have an event scheduled for the very next window. Therefore, each processor determines the next window that has events in it, and the minimum of these is distributed to all processors. After this step, the processor also checks to make sure the endtime hasn't been reached.

After the correct simulation window has been chosen, it in necessary to have each processor distribute the events it has which are scheduled for that window. The events may be for timelines on this processor, or on any other one.

Distributing Events: Let us say that S represents the sub-structure in the event-list that holds the events for the next simulation window. We would like to simply run through all of the events in S, inserting each in its correct destination timeline, and move on. This is possible, because all of the timelines are in shared memory. However, since each processor would attempt to do this at the same time, it is conceivable that two would try to access the same timeline at once.

An alternative would be to send each event to a list on each processor. Once all processors had completed this, each processor could go through this list and insert each event into the proper timeline. This would prevent simultaneous access to timelines. We'd have the same problem again, though, because now two processors could try and write to the same list at the same time.

DaSSF's solution is to have each processor maintain a list for each other processor. Therefore, if there are N processors, Processor P1 has lists L1, L2, ... , LN. When P1 goes through S, if it discovers an event bound for a timeline on P2, it puts the event in L2. In this way, no two processors attempt to access the same list at the same time.

Now, notice that each processor has N lists. If we consider this a row of lists N long, it is possible to stack the rows and create a table, as in Figure 3. The first row of the table consists P1's lists. Remember that L2 is a list of events that are destined for timelines on P2. If we examine the second column of the table, every list in this column contains events for timelines on P2. Thus, once the lists are constructed, P2 can go through every list in the second column and download the events within them to their proper timelines. As P2 will only be interested in column 2, and P1 will only look at lists in column 1, there is no danger of processors attempting to simultaneously access the same list.

Desitination Processor

Source Processor

	Processor 1	Processor 2	Processor 3	Processor 4	Processor 5	Processor 6	Processor 7	Processor 8
Processor 1	L1	L2	L3	L4	L5	L6	L7	L8
Processor 2	L1	L2	L3	L4	L5	L6	L7	L8
Processor 3	L1	L2	L3	L4	L5	L6	L7	L8
Processor 4	L1	L2	L3	L4	L5	L6	L7	L8
Processor 5	L1	L2	L3	L4	L5	L6	L7	L8
Processor 6	L1	L2	L3	L4	L5	L6	L7	L8
Processor 7	L1	L2	L3	L4	L5	L6	L7	L8
Processor 8	L1	L2	L3	L4	L5	L6	L7	L8

Figure 3: Here we see the lists maintained by each processor. They are in a table, showing how events are placed into a list by the source processor and then removed by the destination processor. This is the table for an 8 processors system.

Therefore, distributing events to their proper timelines is a two step process. First, each processor distributes events for the next simulation window into lists, one per destination processor. Each processor pauses until every one has completed this step. Then, each processor downloads the events from all of lists that contain events for that processor into the appropriate timelines.

Activating Timelines: A similar table structure exists for timelines. When a processor takes an event from S (the appropriate sub-structure of the event-list), in addition to putting it in a the correct processor list L, it also determines the timeline that the event is destined for. We'll call this timeline T. T gets placed in a processor specific list, just like the event. In this way, eventually each processor has two lists. One contains events that timelines on this processor will simulate during the next simulation window. The other is a list of timelines that these events are on.

Finally, every processor goes through its list of events and inserts each one onto the appropriate timeline. After this, each one goes through its timeline list. As each timeline on this list has an event during the next simulation window, we need to insure that these timelines are not neglected. Therefore, a dummy process for each timeline is placed on the Process-queue for the appropriate processor. This dummy-process does nothing except suspend instantly. By running it at all, however, control passes to the timeline. By reaching the end of the next simulation window, we guarantee that Process-queue is empty. This means that every dummy Process has been run, so control has been passed to each timeline at least once.

5.1.3 Shared memory dependencies in handle_events()

Clearly, there are three main areas where handle_events() relies on shared memory to function. The first is when the next simulation window is determined. Finding a minimum across all of the participating processors is performed using an array that is N long, where N is the number of processors. Each processor has a spot in the array that corresponds to it and it alone. Each processor's minimum is written into its slot. Then each processor scans the array and copies the smallest element.

The second and third areas are the tables used to distribute events and activate timelines. Clearly, they depend heavily on shared memory. The entire scheme relies on reading data structures created by other processors.

5.1.4 MultiChannels: Shared memory dependencies during simulation

Earlier, we stated that during a simulation window structures created by other processors aren't accessed. This isn't strictly true.

The only events that cross processors are those representing SSF Events. Normally, these are Channel-type KernelEvents. These reference both an SSF Event and an inChannel. Technically, the SSF Event and the KernelEvent represent a usage of shared memory, because these objects may have been constructed by a different processor. However, we consider this as a dependency in the synchronization stage. The inChannel is also not a problem. It is owned by an Entity aligned to a timeline that is on this processor, so referencing it doesn't require shared memory. Therefore, when SSF Events are represented by Channel-type KernelEvents, shared memory isn't necessary - all of the structures accessed are "owned" by the processor running the timeline.

This isn't the case with MultiChannel-type KernelEvents. Earlier, we noted that these reference a _TimeLineTarget object. This _TimeLineTarget is part of the multiChannel that the event was sent from. The Entity the multiChannel is part of may be aligned to another timeline, which in turn may be running on a different processor. This _TimeLineTarget must be accessed by the timeline the MultiChannel-type KernelEvent is on, however. Therefore, multiChannels rely on shared memory.

5.1.5 Shared Memory in Dynamic Load-Balancing

Stealing, the Dynamic Load-Balancing functionality in DaSSF, is heavily dependent on shared memory. When a Process is stolen from one processor, the timeline that Process is on, every Entity aligned to that timeline, and everything owned by those Entities is in effect transferred to the stealing processor. Clearly, this represents one processor accessing objects created by another on a very large scale.

5.2 Dependencies in Model Construction

The steps of Model construction have already been covered in some detail during Section 2. Shared memory is used here to access SSF objects that have been created by different processors. The global naming scheme enables this. An object's memory address can be associated with a character string using addGlobalName(). This name/address pair is been shared with all other processors using resolveGlobalNames(). It is then a simple matter to retrieve this address. Any processor can call getGlobalAddress(), which retrieves the the memory location associated with a character string given as an argument.

In this way, the SSF Model can refer to SSF objects that other processors have constructed. This is essential if the model is going to partitioned among processors. Obviously, this method of referencing objects is completely dependent on shared memory. This problem stems from the SSF API. Whenever an SSF object, such an Entity or an outChannel, is used as an argument to an SSF method, a reference to that object is used. Therefore, when one wishes to align Entity A to Entity B, the argument of alignto() is a pointer to Entity B. The entire API depends upon memory addresses. Thus, if an SSF model is going to be simulated in parallel, shared memory is a necessity. Otherwise, the SSF API has to be extended to recognize the possibility that a pointer to the appropriate SSF object might not be available. We've characterized this dependency as one in model construction because the most SSF methods are called here. The problem of one processor wishing to access something another has constructed arises mainly in two instances. These are channel mapping and Entity alignment.

5.2.1 Shared Memory and Alignment

Generally, all Entities created by a processor stay associated with that processor. This is because they are aligned to Entities on that processor, or are made independent. In either case, the Entity will be simulated by that processor. However, if Entity A created on processor P1 is aligned to Entity B on P2, the responsibility for A is transferred to P2. In the future, P2 will have to access A, which was constructed by P1, constituting a shared memory dependency. Aligning to Entities created by other processors isn't always necessary, and in fact can often be avoided without penalty. Channel mapping is a different matter.

5.2.2 Shared Memory and Channel Mapping

In order to construct a useful model that is distributed among a number of processors, it is essential that mappings between Entities simulated by different processors exist. Without such mappings, every section of the model (and therefore every processor) would be completely independent of all others. Parallel DaSSF would be unnecessary. After all, independent model sections on separate processors are no different from independent models on separate machines. Therefore, we must be able to reference an inChannel on a different processor. This is possible by using global names in DaSSF. Of course, the global naming scheme relies upon shared memory, making channel mapping another of DaSSF's shared memory dependencies.

5.2.3 Shared Memory in other parts on the SSF API

Obviously, alignment and channel mapping aren't the entire SSF API, and we've stated before that the API as a whole depends upon memory references. However, other SSF generally reference objects within the same Entity. A Process waits on its own Entity's inChannels, or writes to its

outChannels. Direct state access from an Entity on one timeline to an Entity on another isn't supported. Therefore, the places where the SSF API seems to require shared memory are generally limited to model construction, during Entity alignment and channel mapping.

5.2.4 Automapping

Automapping is a method by which DaSSF automatically distributes a model among processors. Each new timeline created gets assigned to a different processor. This happens in a cycle, so once every processor has been assigned one timeline, the first processor gets assigned a second timeline, and so forth. The processor that creates the timeline, therefore, often does not ending up running it. Like Stealing, this effectively transfers a timeline and everything associated with it from one processor to another, which is very dependent on shared memory.

6 DaSSF-MPI

DaSSF-MPI permits parallel SSF simulation without shared memory. Instead, data is shared between processors by passing messages. Before we examine the larger issues of implementing DaSSF in a distributed memory environment, we will cover the communications scheme DaSSF-MPI uses.

6.1 Communication in DaSSF-MPI

MPI, the Message Passing Interface, is the API used for inter-processor communication in DaSSF-MPI. MPI provides a great deal of options and flexibility for such communication, but we only exercise a small subset of this. We are primarily interested in three things. These are barrier synchronization, reduction, and passing a message containing data.

MPI_Barrier() performs a barrier synchronization in MPI. This is when all processors pause until they reach a common point. In this way, we can guarantee that no processor will enter part Y of a program until every processor has completed part X.

MPI_AllReduce() is a method of performing a reduction. A reduction is a way to apply some operation to a piece of data on each processor. For instance, if this operation were addition and each processor has some integer i, calling MPI_AllReduce() would return the sum of i's across all processors. Note that this requires the cooperation of all of the other processors. That is, each must call MPI_AllReduce(). Therefore, this also performs a barrier synchronization. Also, MPI_AllReduce() differs from MPI_Reduce() in that the result of the reduction is present on all processors, instead of just one. In DaSSF-MPI, we use MPI_AllReduce() for minimum-finding as well as for sums. If every processor has some value v, by using MPI_AllReduce() we can find the minimum v of any processor.

MPI_Send() and MPI_Recv() are the backbone the API. They are how messages are actually passed between processors. We use them to transfer a number of message types. Mostly, our messages consist of integers or character strings.

Finally, the case often comes up where every processor has to communicate with every other processor. This is like the hand-shaking problem, where there are N people in a room and everyone has to shake hands with everyone else. If we consider that two people shake hands for an amount of time T, and that the time between handshakes is negligible, we wish to minimize the total amount of time spent on hand-shaking. When each processor needs to communicate with every other one,

we use the solution to the hand-shaking problem as a template for communication. That is, the order that people would "shake" is the order that processors communicate with each other.

If there are N people, it can be shown that all the hand-shaking can be done in N*T time if N is even, or (N+1)*T time if N is odd. A method we devised to do this (which we use in DaSSF) is in Appendix A. It should be noted that a similar solution, the Direct Exchange algorithm [9] [10], exists.

6.2 Model Construction

The fundamental problems solved in DaSSF-MPI revolve around the shared memory dependencies covered in Section 5. How can we use SSF with distributed memory? As we stated earlier, SSF methods often require references to SSF objects as arguments. If the SSF object is on another processor, and therefore not in the same memory unit, how can we reference it? In DaSSF-MPI, we use proxies.

6.2.1 Proxies in DaSSF

A proxy object is an object on one processor that represents a second object on another processor. In this way, a modeler can use the proxy as a substitute for the real object, and the DaSSF-MPI run-time system will handle the rest. We can then avoid altering existing parts of the SSF API. We will have to add a few things to it, though.

The basic unit of any SSF model is an Entity. In DaSSF-MPI, we use proxies of Entities to address Entities on other processors. Consider Entities A and B aligned to timelines on processors P1 and P2, respectively. Suppose we wish to construct a channel from A to B. First, we create a proxy Entity on P1 that represents B. Then, we map from A to the proxy. Later, when A attempts to write SSF Events to this channel, the system will deliver these events across the network to Entity B.

We wish to change the modeler's interface to DaSSF as little as possible. To this effect, we've kept the concept of global names. The old scheme, in which character strings were associated with memory addresses, is clearly no longer suitable. Instead of pointing to a chunk of memory, in DaSSF-MPI global names are associated with specific Entities. addGlobalName() has been modified to take a pointer to an Entity and a character string, rather than a general memory address (a void* pointer). Inside the structure that stores global names, or *registry*, information about that Entity is recorded. After resolveGlobalNames() is called, copies on this information reside on every processor. When getGlobalAddress() is used, the information in the registry can be used to construct a proxy Entity. Consider Entity A, which has been added to the registry by globally naming it. pA is a proxy of Entity A. pA is similar to A, except that it has no Processes or children Entities.

In this way, proxies provide us with a logical way to reference objects that don't exist on this processor. Since pA is the same type of object as A, wherever a reference to A should be used as an argument to an SSF method, a reference to pA is legal. Furthermore, from the modeler's perspective, pA doesn't exist. The object returned by getGlobalAddress() can be treated as A.

Now that we've established how the SSF API can remain relatively unaltered, let us move on to considering how proxies actually function. In the build phase, we need to deal with the two areas where DaSSF is dependent on shared memory. These are channel mapping and Entity alignment.

6.2.2 Channel mapping with Proxy Entities

Channel mapping is relatively simple, as we've shown in a previous example. If we wish to map to A to B, we instead map to pB. Remember that A and B must be on different timelines, because a model is partitioned amongst processors by timelines. This means that any SSF Event written from A to B takes at least a simulation window to arrive. Therefore, it will be distributed to the appropriate timeline in handle_events(). Thus, if handle_events() checks if an event is headed for a proxy and re-routes it to the real Entity, everything works fine. The re-routing is non-trivial, because we must encode the event in a message, but we'll discuss this later.

6.2.3 Alignment using Proxy Entities

Entity alignment is a bit more complex. When Entity A aligns to Entity B, it becomes part of Entity B's timeline Y. If Y is on a separate processor, this presents a problem. A proxy timeline pY isn't feasible, because the real Y has to directly access B during simulation. Therefore, if A is on processor P1 and B is on P2, we must transport B to P1. Encoding B into a message isn't trivial, unfortunately. Considering that alignment must take place before an Entity is simulated, another method is possible. We can assume that B is an instance of class C, which is derived from class Entity. If we constructed B2, an Entity of class C using the same arguments in the constructor that P1 used to make B, it would be identical to B. We could then let Y reference this B2, and treat the original B as a proxy (removing B's processes from P2, since P1 is running them). This is DaSSF-MPI's approach.

6.2.4 Automapping in DaSSF-MPI

Above, we've handled two of the major shared memory dependencies in model construction. The third, automapping, is much more difficult. It would require transferring an entire timeline from one processor to another. This timeline and its contents would have to be encoded in messages, which is far from simple. Considering that automapping often results in a worse partitioning than one created explicitly by the modeler, we've decided to leave it unsupported in DaSSF-MPI.

6.2.5 Supporting Proxies in DaSSF-MPI

Proxies, we've shown, are a reasonable way to eliminate the need for shared memory while keeping the SSF API relatively intact. Now we shall look at the details of implementing proxies in DaSSF-MPI.

We've established that a proxy needs to be functionally identical to the real Entity, at least as far as the modeler is concerned. This means it has to be of the same class, and has to have the same general form. For instance, since proxies are used primarily in channel mapping, it is essential that a proxy have the same number of inChannels as the original Entity. How can we accomplish this? Suppose we wish to make a proxy of Entity A on processor P1. We'd like pA to be on P2. Entity A is of class C. If we create a new object of class C on P2, and we give that object's constructor the same arguments that were given to A on P1, then this new object is a reasonable facsimile of A. This is how proxies are created. Of course, to perform all of this, P2 needs to know two things, namely A's class and the arguments given to A. Finally, once P2 know this it still has to create an object of type C.

How can all of this be accomplished? Consider that DaSSF-MPI is only the framework within which a model is simulated. The modeler will almost certainly create their own derived classes from

class Entity. We have no way of finding out about these derived classes without the participation of the modeler. Therefore, we must enlist the modeler's aid, which means extending the SSF API.

6.2.6 SSF API additions to support Proxies

Returning to our previous example, the first thing P2 needs is A's class. To accommodate this, we've added the Entity method getClassname(), which returns a character string. In order to recover the arguments of A's constructor, we've added the Entity methods getArgc() and getArgv(), which return an int and a char**, respectively. It is the modeler's responsibility to represent the arguments given to an Entity's constructor in this fashion.

Even with this information, it still necessary to actually call the constructor for class C. We require some help from the modeler here, as well. The function constructEntity() must be placed within the model. constructEntity() takes the classname, the argc, and the argv which have been gathered, and returns an Entity. This function must be able to create an instance of all the classes derived from class Entity within the model. It should return NULL if any of the arguments don't make sense.

6.2.7 Proxy construction in detail

Generating a proxy is more complicated that giving the correct arguments to the right Entity constructor. If we did this, a full-fledged Entity would result. There are two important areas where a proxy differs from a real Entity. The first is that a proxy has no processes. The second is that constructing a proxy shouldn't result in sub-Entities. The first is simple to implement. Once a proxy Entity is created, DaSSF-MPI explicitly deletes each of its processes. The second difficulty is little tricker, though. We wish to prevent any new Entities from being created during the proxy's constructor or init() method. In order to accomplish this, we basically set a flag that restricts DaSSF from creating any new Entities. We are careful to unset this flag immediately after the proxy is dealt with. However, this requires some diligence on the part of the modeler. A check must be made before referencing a newly created object, to make sure it exists. For instance, let us suppose the init() method of class C creates and accesses Entity B. If a proxy of an object of class C were made, the call to create Entity B would accomplish nothing and return a NULL pointer. The subsequent attempt to access B will surely cause an error.

6.2.8 Alignment to Proxies in detail

Aligning an Entity to a proxy is handled in a similar fashion to creating a proxy. We shall examine this by aligning Entity B to proxy pA. A is the Entity that pA represents. A is aligned to timeline X on processor P1, while B has been created on P2. In order to align B to A, we use constructEntity() to create a facsimile of B on P1, which we will call B2. Like a normal proxy, we don't want B2 to create sub-Entities, so we restrict this. B2 is expected to act like a real Entity, though. Therefore, we leave B2's Processes intact. After B2 is constructed, it effectively becomes the "real" B. Therefore, we reduce the original B to a proxy of B2. This conversion is accomplished by removing all of B's Processes. We leave Entities created by B intact, however.

That addresses the shared memory concerns associated with model construction, which were channel mapping, Entity alignment, and timeline automapping. Now, let us consider the dependencies encountered while the simulator is running.

6.3 Model Simulation

The primary use of shared memory once the simulator is running happens during handle_events().

6.3.1 Choosing the next Simulation Window

The first area in handle_events() where shared memory is required comes when determining the next simulation window. Each processor chooses its own minimum window, and the smallest of these is chosen. This presents no problem without shared memory. If each processor has a value, and we wish to find the minimum of these values, we perform a minimum reduction. This is built in to MPI, in the form of MPI_AllReduce().

6.3.2 Distributing Events

KernelEvents are distributed from each processor's event-list to the appropriate processor via a table structure covered in section 6. Obviously, this method doesn't work without shared memory.

The problem isn't simply that the table can't be created, however. Without shared memory, moving a KernelEvent from one processor to another requires encoding it in a message. This must be addressed before larger issues can be considered.

Encoding an object into a message requires a record of everything in that object. Certain things, such as integers and character strings, are easy to encode. Other items, such as memory references, are problematic. We solve this by explicitly encoding the parts of a KernelEvent that aren't pointers in a message, sending the message, and "reattaching" the pointers to equivalent structures on the destination processor.

Encoding a KernelEvent: The only KernelEvents that cross processors are of types Channel or MultiChannel. We will concern ourselves with Channel-type ones here. A Channel-type KernelEvent has two pointers, one to an SSF Event and one to an inChannel. Obviously, no equivalent SSF Event exists on the destination processor. However, an SSF Event is easy to encode in a message. Therefore, we encode the SSF Event as part of the encoded KernelEvent. The inChannel, on the other hand, does exist on the destination processor. As this event is crossing processors, we know the destination inChannel is owned by a globally named Entity in the registry. Consider that KernelEvent e represents SSF Event E which is traveling from Entity A to Entity B. A in on processor P1, B in on P2. This means that A's outChannel is actually mapped to proxy pB on P1. By looking at which inChannel on pB which A maps to, we can locate the equivalent "real" inChannel on B. When it was on P1, e pointed to an inChannel on pB. Once e in encoded and sent as a message to P2, its pointer is attached to the equivalent inChannel on B. In this way, it is possible to transport a Channel-type KernelEvent from one processor to another using message passing.

Encoding an SSF Event: Transferring KernelEvents depends on the fact that SSF Events are easy to encode. The modeler, however, may be sending instances of custom classes that are derived from SSF Events. If this is the case, we make encoding instances of these new classes the modeler's responsibility. A method similar to the one used for encoding Entities is used. SSF Events have been extended to have getClassname(), getArgv(), and getArgc() methods. A constructEvent() function also has to be provided by the modeler. There is one difference between the handling of Events and Entities, however. getArgv() and getArgc() should represent the Event's current state. Consider if getClassname(), getArgv(), and getArgc() are called on Event E, and then used

as arguments for constructEvent(). The new Event F that constructEvent() returns should be identical to E. That means that F and E should have the same internal state. This allows us to transfer SSF Events between processors using message passing.

Encoding and Distributing KernelEvents: Now we can return to the method of using a table to distribute events. This is obviously not viable without shared memory. However, the first step can still work. That is, a processor maintains a separate list for each processor, and places the events it wishes to distribute in each list accordingly. It is the second part, where some other processor accesses these lists, that is no longer feasible. Instead, we use the hand-shaking algorithm covered earlier. Each processor communicates with each other processor. Consider processors P1 and P2. P1 has a list L2 which contains KernelEvents that should be on P2. P2 has a corresponding list L1 which contains KernelEvents for P1. When the hand-shaking algorithm dictates that it is time for P1 and P2 to communicate, they exchange lists. First, P1 encodes each KernelEvent in L2 into a message and sends the message to P2. Then, P2 encodes everything in L1 and sends it to P1. Once every processor has communicated to every other one, each processor has all of the KernelEvents that it would have received using the table method.

6.3.3 The timeline Table

Recall that a similar table method was used to note timelines that would be active in the next simulation window. We can't use the same approach here that we used with KernelEvents. This is because encoding a timeline in a message is so complex that we didn't attempt it. However, the point of this step is to note which timelines are going to be active. This is done by examining the KernelEvents as they are removed from the event-list and distributed to the appropriate processor. Discovering the timeline that each event is destined for can be done just as easily after event distribution as before. After distribution, we know that a KernelEvent and the timeline it is bound for are on the same processor. By delaying the step where we find active timelines until after KernelEvent distribution, we obviate the need for shared memory.

6.3.4 MultiChannel-type KernelEvents

In the proceeding few sub-sections we demonstrated a method that allowed Channel-type KernelEvents to be transferred amongst processors using message passing. Unfortunately, MultiChannel-type KernelEvents are not as simple. Instead of referencing an inChannel, they point to a _TimeLineTarget object, which we will refer to as a tlt. This tlt is needed to allow the destination timeline to process the KernelEvent. The tlt is part of the multiChannel, though, which is on the source, not destination, processor.

One option would be to encode the tlt and send it with every MultiChannel-type KernelEvent. This is feasible, because everything the tlt references has a corresponding structure on the destination processor. However, an encoded version of a tlt isn't necessarily very small. Furthermore, the whole point of a multiChannel is that it's more efficient than a collection of normal outChannels. By encoding the tlt with each MultiChannel-type KernelEvent, we'd make using a multiChannel expensive, which defeats the whole point of having a separate multiChannel class.

Instead, we ensure that an equivalent tlt already exists on the destination processor. We accomplish this during model construction. When a multiChannel is created, it begins to keep track of its input. Whenever it is mapped to an inChannel, it makes a record of these. These records are placed into separate lists. Each list corresponds to a separate processor. A record of

the mapping to an inChannel is placed in the list representing the processor that the inChannel is on. After the build phase, all of the channel mappings are in place. Then, each processor talks to all of the others. If a multiChannel on P1 has been mapped to some inChannels on P2, a copy of that multiChannel is made on P2. This copy is mapped to each of the inChannels on P2 that the original was mapped to. This way, the new multiChannel on P2 has identical tlts as the original for the timelines on P2. Then, when MultiChannel-type KernelEvents from the old multiChannel arrive on P2, they can be set to point to a tlt on the new multiChannel, which is equivalent to the tlt they used to point to on P1.

6.3.5 Timeline Stealing

Dynamic Load-Balancing, in the form of Process stealing, requires transferring a timeline from one processor to another. This is easy in normal DaSSF, but requires shared memory. As we've noted earlier, we consider the problem of transferring a timeline using message passing extremely difficult, and we haven't attempted to solve it. As such, there is no stealing in DaSSF-MPI. Even if one were to transfer a timeline via message passing, however, such a transfer would be expensive. A model would have to be seriously imbalanced to make this load-balancing worth the cost. If a model is reasonably well-partitioned by the modeler, it is doubtful that dynamic load-balancing would be useful, even if it were an option.

6.4 Model Construction revisited

Much that is accomplished during the model construction phase requires message passing. Global names are distributed, so that each processor has an identical registry. Entities may need to be aligned to off-processor timelines, which requires letting the destination processor know that a new Entity must be constructed. Information regarding multiChannel mappings must be exchanged, so that new multiChannels with corresponding tlts can be created on the appropriate processors. In order to better understand all of this, we will examine what has been changed in the root Process. The changes take place just before start_all() is called on the initialized Entities. It assumed that resolveGlobalNames() was called by the model at this point. After all, channel mappings take place in the build() phase, which has already completed. In order to map channels between Entities on different processors, resolveGlobalNames() must be called.

Off-processor alignments: We've just established that resolveGlobalNames() has been called by the completion on the build phase. If an Entity have been aligned to off-processor timelines, this means the processor which that Entity resides upon has changed. One of the pieces of information the registry holds about each globally named Entity is the processor it is on. In order to align an Entity to another one off-processor, both must be globally named. Therefore, by aligning the Entity, the information in the registry about what processor that Entity is on becomes outdated. In order to correct this, a round of communications takes place where each processor's registry is updated to reflect the off-processor alignment.

Channel mapping and off-processor alignments: In cases where an Entity is aligned off-processor, the original processor is responsible for mapping that Entity's channels. However, all that is left on the original processor is a proxy, and mapping a proxy's channels does not effect the real Entity. Therefore, a record of the mappings is made. This record is then passed to the destination processor, so that the mappings can be applied to the real Entity. This exchange of

records requires inter-processor communications between all processors, which takes place at this point in the root Process.

MultiChannel support: After this, we go through the necessary steps to support multiChannels, which have been outlined above. This also involves all processors exchanging information.

6.5 DaSSF-MPI limitations and usage changes

Users of DaSSF-MPI should be aware of a number of things. If a modeler wishes to align an Entity to a timeline on a different processor, everything relating to that Entity must be globally named. That is, that Entity, the one it is getting aligned to, and everything the first Entity is mapped to must be in the registry. The need for this can be seen by examining the modifications to the root Process.

6.5.1 New Entities during Simulation

The next item that DaSSF-MPI modelers should keep in mind is that all important Entities should be created before simulation starts. That is, an Entity created after simulation time starts has very limited utility. First of all, it can't be globally named, because resolveGlobalNames() has already been called. This means it cannot be mapped to by Entities on other processors. Likewise, it cannot be aligned to these Entities. Finally, it cannot contain any multiChannels.

The problems with mappings and alignment could be overcome if the registry could be modified after resolveGlobalNames(). However, this would require updating all of the registries. That would only be convenient at simulation window boundaries, which would restrict when a new Entity could be created.

The problems with multiChannels wouldn't be solved by a dynamic registry. Instead, new multiChannels could be supported by encoding a tlt in a MultiChannel-type KernelEvent the first time that tlt was accessed. This would take place instead of the steps used in the root Process to support multiChannels. The receiving processor could remember the tlt that came with the MultiChannel-type KernelEvent, so subsequent KernelEvents that wish to access that tlt wouldn't have to carry an encoded copy. This is arguably a more robust solution. However, due to global naming issues noted above, useful Entities basically must be created before simulation time starts. Since this is the case, the multiChannel support used in DaSSF-MPI is sufficient. Finally, it is rarely useful to create Entities after time begins to advance, so these aren't major shortcomings.

6.5.2 Static Alignment and Channel Mapping

Another obvious limitation in DaSSF-MPI is that Entities and channel mappings are static. They can only be set once, and can't be changed during the simulation. However, this is also true of DaSSF 1.22, which DaSSF-MPI is based on. Therefore, these limitations are to be expected.

6.5.3 Usage Changes

Finally, there are a few usage changes in DaSSF-MPI, as compared to DaSSF 1.22. The options -steal, -stealevel, -automap, and -backstore are useless, because these options are unsupported. The -nprocs option is also unused. This is because the addition of MPI means the program mpirun must be used to start DaSSF. One of the command line arguments to mpirun in -np, which specifies the number of processors. This replaces -nprocs.

7 DaSSF using Shared and Distributed Memory

We would like DaSSF to exploit parallelism in a distributed memory environment principally for two reasons. The first, to enable parallel DaSSF simulations on more architectures, is accomplished by DaSSF-MPI. The second goal is to have DaSSF operate in parallel using both shared and distributed memory. For example, suppose we have two computers with 4 processors each, and each computer shares memory between its processors. Ultimately, we would like to distribute a model across all 8 processors. DaSSF-MPI is only an intermediate step towards this goal. For ease of reference, we'll call this shared+distributed memory version of DaSSF as DaSSF+. DaSSF+ represents future work. It is what we'd like to accomplish with the lessons learned while implementing DaSSF-MPI.

7.1 DaSSF+ in operation

Lets consider how distributed memory and shared memory might be used within one system. Obviously, shared memory access will always be faster than passing messages over a network. Therefore, we will use two simulation windows. One, the smaller of the two, will be for synchronizing processors sharing memory. The second, larger window will be used for distributed memory synchronization.

This is easier to understand with an example. Let's say that we have two machines, each with a number of processors that share memory. Suppose we have a model of two subnetworks connected to each other using some sort of slow interconnect. Two LANs connected over the Internet, for instance. Now, if each computer on a LAN is represented in the model by an Entity, the network connections would be represented as channels between them. As each computer on a LAN can operate independently of the others, each Entity would be on its own timeline. This can be seen in figure 4.

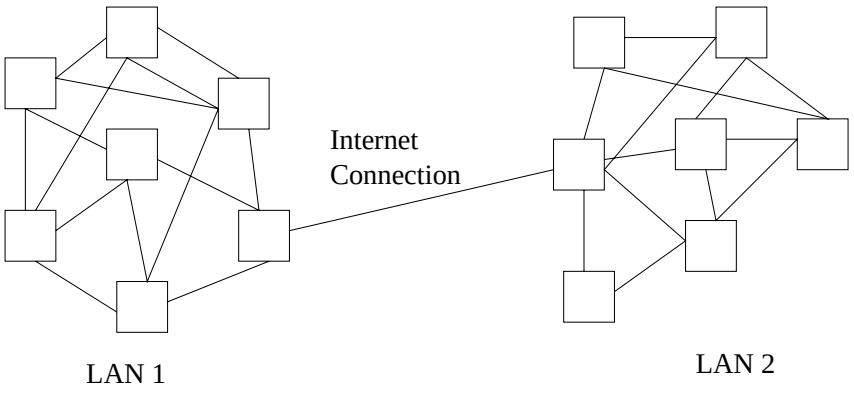

Figure 4: A Torus of Processors.

The minimum cross-timeline channel delay, which we will call MD2, would exist between two computers on the same LAN. This is because the intraLAN interconnect is the fastest present. Therefore, it would be possible to run this model with a simulation window of size MD2. At each window boundary, all of the processors which share memory would distribute events in the same manner DaSSF 1.22 does. Processors wishing to send events to processors in the other machine could pursue the strategy DaSSF-MPI uses. This approach would be sufficient to accurately simulate the model. It would also incur the cost of message-passing at every simulation window

boundary, however. If this cost were great enough, using DaSSF+ might actually be slower than DaSSF 1.22.

What would happen if we distributed the model so that all the Entities representing the computers in LAN 1 are on the first machine, and everything in LAN 2 is on the second? This would mean that the minimum delay for events going from machine one to machine two would be determined by the speed on the connection between the LANs. We've stated earlier that this is slower than the interconnect within each LAN. Let's call the minimum delay of channels between the LANs MD1. Events crossing between machines are distributed using message passing, but those that stay within processors on one machine take advantage of shared memory. This means that a larger simulation window, one of size MD1, can be used for events that use message passing. Therefore, a simulation of size MD2 can be used for intra-LAN events. Then, we only have to use message passing, which is relatively expensive, for inter-LAN events exchanged at the boundaries of windows which are of size MD1.

7.1.1 DaSSF+ with two simulation windows

Let W1 be the simulation window representing the amount of time we can run models on each machine independently before synchronizing the machines. Let W2 be the window that represents the amount of time processors within one machine can be run before they need to synchronize with each other. In a well partitioned model, W1 should be larger that W2. We can use DaSSF-style synchronization (which uses shared memory) between processors on one machine at each W2 boundary. The kind of synchronization in DaSSF-MPI will only be performed at W1 boundaries. In this way, we only incur the cost of message passing when it is absolutely necessary to insure proper simulation.

7.2 Full SSF compliance with distributed memory

The two-window approach to synchronization is sufficient to integrate DaSSF 1.22 and DaSSF-MPI. A DaSSF-MPI-style global naming scheme would function adequately in DaSSF. DaSSF-MPI enforces a number of limitations on an SSF model, though. For instance, all useful Entities must be created before simulation time begins to advance. In addition to this, channel mappings are static, as is alignment. These limitations are present in DaSSF 1.22 as well. DaSSF 1.22 was the current version of DaSSF when DaSSF-MPI development began. Since that time, though, DaSSF has advanced to the point where it is fully SSF compliant, so the above restrictions are no longer present. In order to create DaSSF+, we must also support this functionality with distributed memory.

7.2.1 New Entities during simulation

In DaSSF-MPI, it is difficult to make an Entity of much use during simulation. This is because the registry is frozen after resolveGlobalNames(), and also due to the way multiChannels support is implemented. This must be remedied in DaSSF+. In order to have a dynamic registry, changes to it on one processor must be reflected on all others. This requires some synchronization. It would be difficult to perform this in the middle of a simulation window. Fortunately, the SSF specification allows us to fulfill requests from the modeler at our leisure. Therefore, when a new Entity is created, the modeler cannot make any assumptions about that Entity until a time specified by the framework. We can set this time as the next simulation window boundary. More specifically, this

is the W1 boundary. This will allow us to update the registries on all processors, which gives us a dynamic registry. That will let us create new Entities during simulation.

The multiChannel support can be handled similarly. Following a "registry update" phase in synchronization, we can have a "multiChannel update" phase as well. If a new multiChannel were created, in this phase we could create tlts on the appropriate processors to support this multiChannel.

7.2.2 Dynamic channel mapping

Support of dynamic channel mapping does not present any great obstacles. Again, SSF allows the framework to specify when the requested channel mapping takes effect. Therefore, new channel mappings will not be valid until a W1 boundary has passed. This is because a mapping change would have to be reflected to other processors.

For instance, let us consider Entity A, which has multiChannel mC and is aligned to timeline X on processor P1. Meanwhile, Entity B (which is aligned to Y on P2) has inChannel iC. what if mC becomes mapped to iC once the simulation is running? Perhaps mC has already been mapped to some other inChannel aligned to Y. This means that a tlt for Y already exists on P2. This tlt must be updated to reflect the new mapping. The update would take place during synchronization at a W1 boundary.

We can therefore support dynamic channel mapping if we can don't let the mapping take effect until a W1 boundary. Since SSF allows us to impose this restriction, allowing such mappings in DaSSF+ should not be a problem.

7.2.3 Dynamic Entity alignment

Dynamic Entity alignment is more complicated than dynamic channel mapping. In terms of keeping the registry up to date and updating structures on each processor, the same techniques used for channel mapping apply here. That is, if we say that a request for an Entity realignment isn't fulfilled until a W1 boundary, then keeping that Entity's proxies, etc., on other processors updated isn't a problem.

A greater difficulty exists in actually realigning the Entity. Specifically, what happens when we align Entity A to Entity B, when A is on processor P1 and B in on P2, and there is no shared memory between P1 and P2? We would like to encode A in a message and send it to P2. Unfortunately, it is much more difficult to encode an Entity than an Event. The main problem is A's Processes, which are all currently running on P1. Moving them means saving their state and the point at which they are suspended, and then recreating them from this information on P2. This is not a simple task. In fact, encoding an Entity is probably the largest obstacle to creating DaSSF+.

7.3 Conclusions about DaSSF+

DaSSF+ would allow us to reap the benefits of parallel simulation in shared and distributed memory environments. Many of the techniques used to implement DaSSF-MPI could be similarly applied in DaSSF+. Some of them would require modification to be fully SSF-compliant. Most of these modifications should present no difficulties. Supporting dynamic Entity realignment, however, is complicated. Solving this problem well will be the primary challenge in creating DaSSF+.

8 DaSSF-MPI Performance

In order to see how well DaSSF-MPI performs, we created a sample SSF model and simulated it under a number of different conditions. We used DaSSF 1.22, DaSSF-MPI for Irix, and DaSSF-MPI for Linux. The test machine for DaSSF 1.22 and DaSSF-MPI for Irix was an SGI Origin 2000. DaSSF-MPI for Linux was run on a network of 4 Gateway 2000 Intel-based machines connected with a Myrinet switch. The full specifications for the test machines can be found in Appendix B.

8.1 The Model

Before going into the test results, we will briefly cover the SSF model used. We simulated traffic on a torus of processors, each connected to their neighbors.

8.1.1 The Torus

Assume we have a grid of processors. Each processor is connected to four others. If we imagine the processors are in a two-dimensional array, then one processor is connected to the others to its left, right, above it, and below it. This creates a large rectangle of processors. If we then wrap the rectangle into a hoop, connecting the processors on the "hoop boundary", we've created a torus. This can be seen in figure 5.

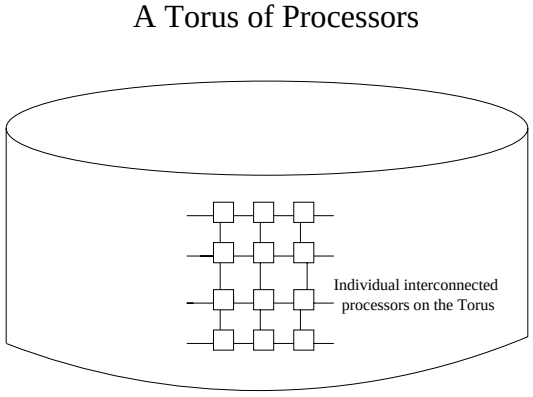

Figure 5: A Torus of Processors.

Now, assume processor A wishes to send a message to processor B. In this type of network, that message has to travel through a string of intervening processors. Therefore, each message has an address. When message M arrive on processor P, P examines M's address and forwards it in the proper direction.

8.1.2 Representing a Torus of Processors in SSF

This network of processors translates rather easily into an SSF Model. Each processor is an Entity. That Entity has four outChannels and one inChannel. Each outChannel corresponds to a neighboring processor. All of processor P's neighbors map their appropriate outChannel to P's inChannel. For instance, let's say P s left neighbor is lP. lP has an outChannel leaving to it's right. This outChannel would be mapped to P's inChannel.

One possible way to simulate this torus requires each Entity to have two Processes. One, a traffic generator, would randomly choose a destination processor, and then send a random amount of messages to that processor. The second Process would wait on the inChannel. When a new message arrives, it would examine it and forward it to the appropriate neighbor.

In order to more accurately simulate this network, though, we should allow for dropped messages. We assume that a processor can only hold a set amount of messages internally, presumably in a queue. If it receives too many messages before it can empty this queue, then newly arriving messages are lost. Therefore, we actually have three Processes. The first is the traffic generator. The second puts arriving messages in the queue, or discards them if the queue is full. The third removes messages from the queue and forwards them to the appropriate neighbor.

As for the messages themselves, they are SSF Events. Actually, as they need to carry an address with them, they are really instances of a class derived from SSF Events. This derived class contains routing information, so the message can reach its destination.

8.1.3 Partitioning

The model is partitioned into subsections of the torus. Each partition is a rectangular "slice" of the torus. Each processor is responsible for a slice that is as nearly equal in size to the other slices as possible. An example of this is in figure 6.

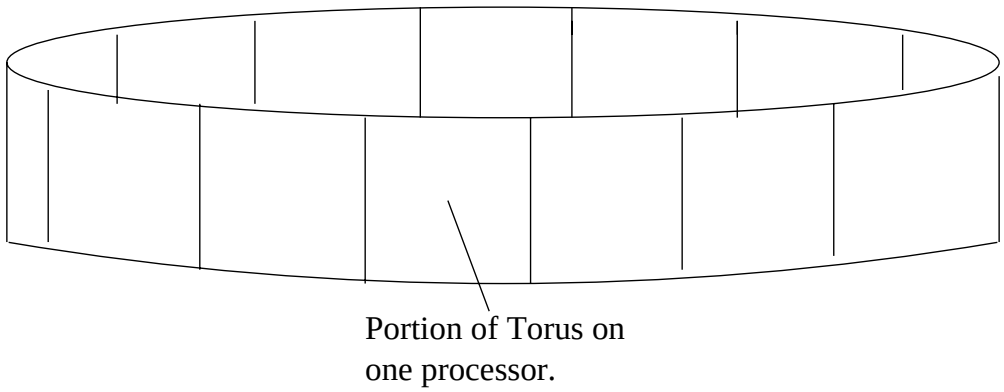

Figure 6: A Torus partitioned onto 12 Processors

8.2 Testing and results

Before we get into the specific tests, we'll cover a bit of the general methodology. Each data point is the average of ten samples. Therefore, when we say that DaSSF-MPI for Linux takes X seconds to simulate a 10x40 torus on 4 processors until time 10000, we've ran this configuration 10 times and taken the average running time. The error bars in the graph represent the range of the test results at each data point.

8.3 Applying more Processors

The first test we applied examines the benefits of parallelism. Here, we successively applied more processors to a constant size model. In each of these tests, we used a 10x40 size torus. A sample

10x40 torus can be seen in figure 7.

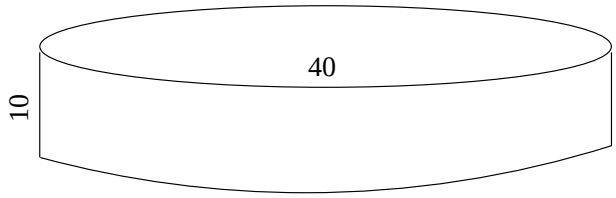

Figure 7: A 10x40 Torus: 10 processors high, 40 processors around.

We simulated this torus until time 10000. We then scaled the number of processors from 1 up to 4 (the maximum number we had on the network of Linux machines). The results are in Figure 8.

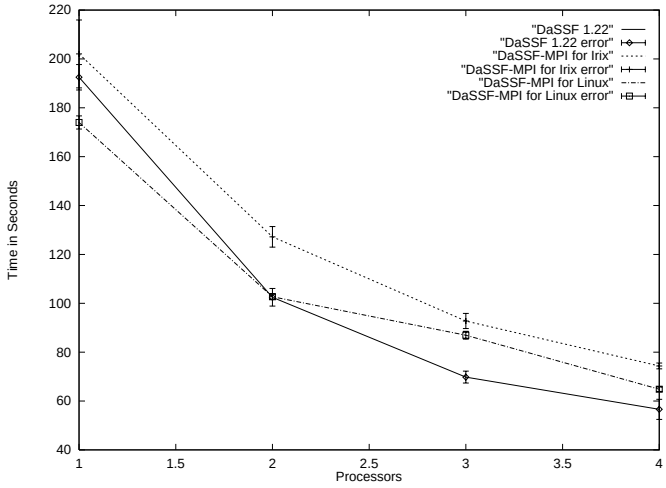

Figure 8: Constant Model Size, changing Processors

For the size of the model used here and the number of processors applied, DaSSF-MPI applies extra processors about as well as DaSSF 1.22. As we will see below, the greater costs of synchronization in DaSSF-MPI play a significant role when larger number of processors are used.

8.4 An increasing Model Size

In this test, we kept the number of processors used constant. Instead, we increased the model size. Four separate models were used, of size 10x20, 10x40, 10x60, and 10x80. Each model was simulated until time 3000. The results are in Figure 9.

DaSSF-MPI fares well here. DaSSF-MPI for Linux doesn't deal with the larger models as well as the versions for Irix. This is probably caused at least in part by the relative slowness of Myrinet. A larger model results in more traffic, which means more SSF Events that must be transferred over the network. The network connecting the Linux machines is certainly slower than the connections between processors inside the Origin 2000.

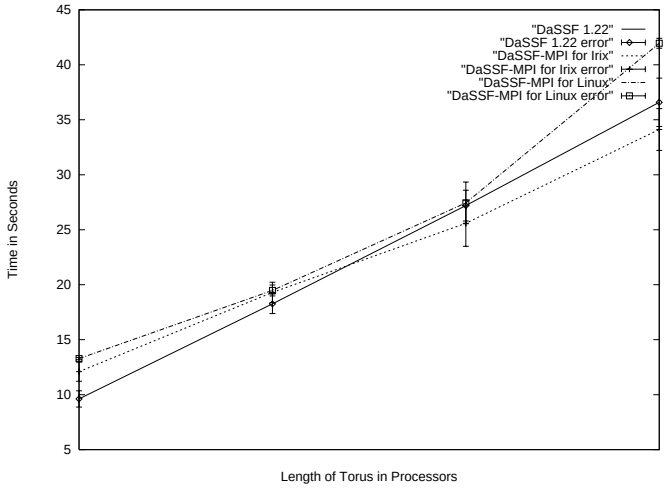

Figure 9: Changing Model Size, constant Processors

8.5 Scaling the Model and the Processors

Here, we increased both the model size and the number of processors used. First we used a 10x10 torus on 1 processor. Then we tried a 10x20 torus on 2 processors. We continued this up to a 10x40 torus on 4 processors, always insuring each processor was allocated a 10x10 grid to simulate. Each model was run until time 10000. The results are in Figure 9.

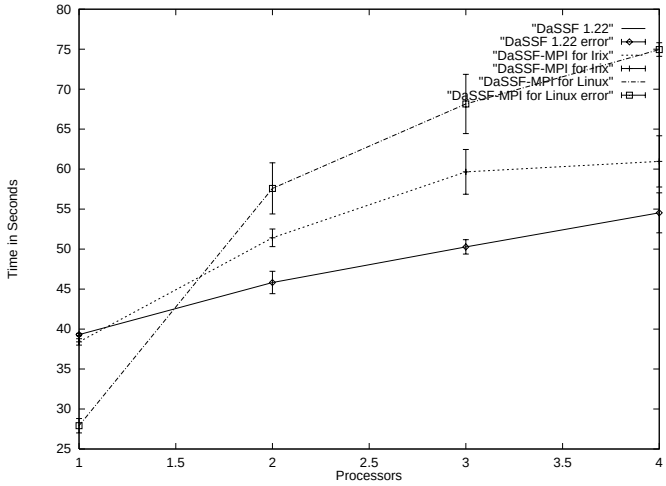

Figure 10: Scaling Model Size and Processors

DaSSF-MPI for Irix apparently doesn't scale as well as DaSSF 1.22. It does, however, consistently stay less than 20% worse than DaSSF 1.22. These results probably reflect the overhead of message passing. It should be noted that although the grid size per processor stays constant, the events per processor grow. This is because increasing the grid size causes more traffic at a greater

than linear rate. Therefore, doubling the grid more than doubles the traffic, which results in more messages using MPI, which in turn slows DaSSF-MPI relative to DaSSF 1.22. Considering that DaSSF-MPI is slower by a constant factor or less, we consider these results reasonably good. As for DaSSF-MPI for Linux, here we see the penalties of Myrinet easily. Initially, when the network isn't used at all, the model runs very quickly. Once the network is in use, though, it rapidly slows down. Its behavior follows the same general trend as DaSSF-MPI for Irix, which is to be expected.

8.6 DaSSF-MPI vs. DaSSF 1.22: more Processors

In addition to the previous results, we also examined how DaSSF-MPI compared to DaSSF 1.22 when more processors were involved. For these test, we used a somewhat faster Origin 2000 than in the previous ones. Here, we ran each test with up to 8 processors. As our Linux network had only 4 machines, we did not include DaSSF-MPI for Linux in these tests. The specifications of the Origin 2000 used for these tests can be found in Appendix B.

8.6.1 Applying more Processors

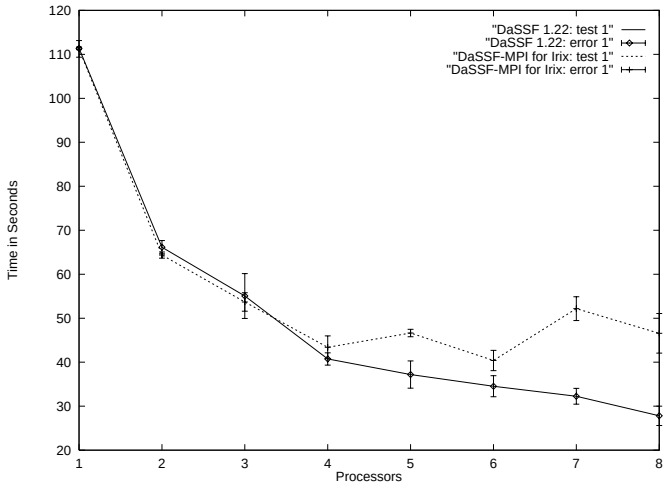

Figure 11: Constant Model Size, changing Processors

As can be seen in Figure 11, applying more processors to a model doesn't neccessarily result in a performance gain in DaSSF-MPI. This is because of the larger cost of synchronization associated with distributed memory. The cost of sychronization grows approximately linearly with the number of processors used in both DaSSF-MPI and DaSSF 1.22, but the constant factor is much larger in DaSSF-MPI. What we see in Figure 11 is the costs of synchronization outweighing the benefits of more parallelism in DaSSF-MPI. It is faster to simulate this model on 4 processors than on 8. If we were to use a bigger model, so that each processor were responsible for simulating a larger part, then the opposite would be the case. This is because the simulating the model in parallel would save more time than is lost in slower synchronization phases.

The "choppiness" of the DaSSF-MPI graph is also interesting. The communication scheme used in DaSSF-MPI takes roughly the same amount of time for N processors as for N+1, if N is odd. That is why applying 5 processors is slower than applying 6, but faster than using 7. The

syncrhonization costs are roughly the same for 5 processors as for 6. Each processor is responsible for a smaller part of the model when 6 processors are used, though, which explains the performance gain of using 6 vs. 5.

8.7 An increasing Model Size

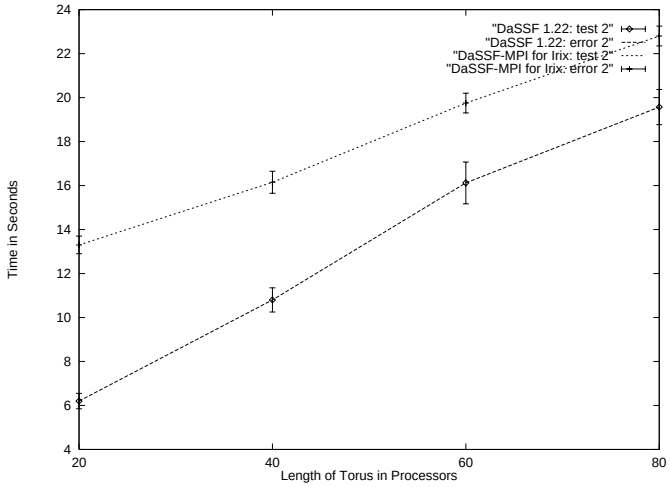

Figure 12: Changing Model Size, constant Processors

Figure 12 shows that DaSSF-MPI fares well with larger models. These results are in accordance with those presented earlier, in sub-section 8.4. As we can see, DaSSF-MPI is slower than DaSSF 1.22. However, the slope of DaSSF-MPI's performance is less than DaSSF 1.22. This happens because the added load on each processor is reducing the effect synchronization cost has on performance in DaSSF-MPI. This synchronization cost is less of a factor in DaSSF 1.22, so added load does not change it's effects on total performance as much, which results in a greater slope. We postulate that if this graph were extended, DaSSF-MPI's slope would change to approach DaSSF 1.22's, but the lines would not cross.

8.8 Scaling the Model and the Processors

The results of scaling the model size and the number of processors proportionately can be seen in Figure 13. DaSSF-MPI performs about as well as DaSSF 1.22 for even numbers of processors. The costs of the DaSSF-MPI synchronization method can be seen when odd processors are used, though.

8.9 Results of testing

DaSSF-MPI compares well with DaSSF 1.22. In most cases it performs almost as well, and at worst it is slower by a constant factor. A very interesting thing to note is that DaSSF-MPI for Linux normally does fairly well compared to DaSSF 1.22. Considering that a network of Linux machines is much less expensive that an Origin 2000, this suggests that DaSSF-MPI for Linux is a more economical way to run SSF simulations. It should be noted that DaSSF 1.22 is compiled

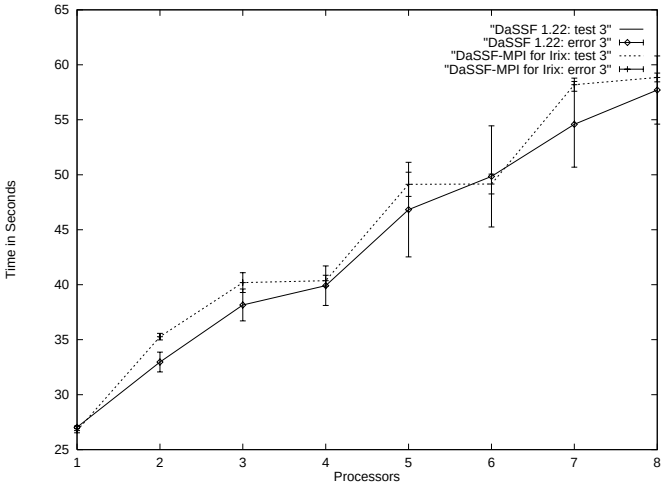

Figure 13: Scaling Model Size and Processors

with the gcc compiler. Later versions of DaSSF use the native SGI compiler on the Origin 2000. This has resulted in a reported speedup of around 200%. Thus, although running DaSSF-MPI on a Linux network is certainly cheaper than using DaSSF on an Origin 2000, the performance cost is larger than the above results imply. We felt that testing with DaSSF 1.22 is justified, as this is the code-base that DaSSF-MPI is derived from.

As an interesting side note, the above tests also demonstrate the cost of message-passing vs. shared memory on the Origin 2000. While a performance hit does exist, it is surprisingly small in most cases. As we've noted earlier, the worst results we encountered were on the order of 20%, and they were often much less.

Finally, it is clear from the testing that synchronization costs do not scale as well in DaSSF-MPI as in DaSSF 1.22. This is to be expected. It means that applying more processors in DaSSF-MPI is only useful when the per-processor modelling load is significant.

9 Future work

The next logical step is to add distributed memory functionality to the current version of DaSSF. This will create DaSSF+. We will begin work on DaSSF+ in the Fall semester of 1999. MPI will probably be discarded in favor of HLA RTI (the High Level Architecture Run-Time Infrastructure). This is a package that provides a number of simulation services. Among these services are synchronization and messaging. We briefly explored HLA RTI while attempting to extend DaSSF-MPI to a TCP/IP network. This version would have used two simulation windows, as we proposed for DaSSF+. The smaller would have represented machines with fast interconnect, such as Myrinet. The second would be for the slower TCP/IP connections. Unfortunately, due to time constraints we were unable to implement this. While attempting to, though, we examined HLA RTI and will probably use it in DaSSF+. We hope to have DaSSF+ completed by late 1999 or early 2000.

10 Conclusions

DaSSF provides an implementation of the SSF specification. One of the strengths of DaSSF is the performance it achieves through parallelism. Enabling this feature on more architectures increases the usefulness of DaSSF. Many architectures don't have multiple processors that share memory. This means that parallel processing on these architectures must use distributed memory. DaSSF has to be rather fundamentally altered to accommodate this.

The SSF specification itself would seem to restrict us to shared memory parallelism. It can be minimally extended, though, to support distributed memory. We have extended it in such a manner.

DaSSF-MPI represents an implementation of DaSSF that achieves parallel simulation without depending upon shared memory. It also achieves our goal of extending parallel DaSSF to new architectures by enabling it on Linux.

DaSSF-MPI also explores techniques for general distributed memory use in DaSSF. These techniques are not dependent upon MPI, so other communication packages could be used. The methods in DaSSF-MPI can be extended to provide full SSF compliance. By allowing DaSSF to use both shared and distributed memory, we could greatly enhance its functionality.

By porting DaSSF-MPI to Irix, we provided a method to compare MPI with shared memory. Although this was not one of our primary goals, our results suggest that the cost of MPI is surprisingly small.

Finally, DaSSF-MPI has met all of the goals we had for it when we began this thesis. It extends parallel DaSSF to Linux without an undue performance penalty. In addition, it has illustrated the issues that need to be dealt with in DaSSF+. We consider it a success on all fronts.

11 Appendix A: the hand-shaking Algorithm

The hand-shaking problem is the same as the following: how quickly can we make all of the possible connections between the vertices of an n-sided polygon? We assume at each step, we can only connect one vertex to one other vertex. Thus, at step S, if we connect vertex i to vertex j, we cannot connect i or j to any other vertex in that step. In this way, each connection represents a handshake and each step takes the time of a handshake.

If the polygon has N vertices, then a total of $\sum_{1..N} i$ connections exist. Lets consider an odd polygon. A total of $\frac{N-1}{2}$ connections can be made in one step. This means the optimal time is equal to $\frac{\sum i}{\frac{N-1}{2}}$ steps, rounded up. Let us define the length of a connection as the least number of intervening vertices between the two vertices being connected. Therefore, connecting two adjacent vertices results in a 0 length connection. In any odd polygon, connections of length 0 to length $\frac{N-1}{2}$-1 are made. Furthermore, N connections of each length are made. Let us examine this when N = 5. Figure 10 shows that there a 5 connections of length 1 and 5 connections of length 0.

If in each step we can insure that $\frac{N-1}{2}$ new connections are made, that is sufficient to reach the optimal solution. This is easy to accomplish. If we attach two adjacent vertices, and then the two vertices adjacent from them, and so forth, we can create 1 edge of each possible length in one step. If we then choose two unconnected adjacent vertices in the next step, it can be seen by inspection that each step will create new connections (i.e., each connection will only be made once). The connections that would be made in a single step on a 5-vertex polygon can be seen in Figure 14.

In this way, we can create every connection in $\frac{\sum i}{\frac{N-1}{2}}$ steps (rounded up). That means that this

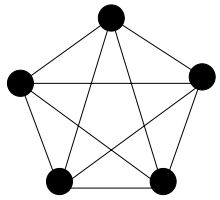

Figure 14: A fully connected 5-vertex polygon

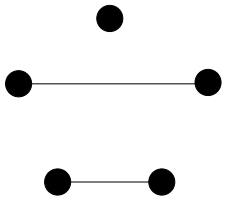

Figure 15: Connections made in 1 step

method provides an optimal solution to the hand-shaking problem for an odd number of people.

This solution can be generalized to even values of N in the following way. Note that when N is odd, each step has one vertex which is not involved in any of the new connections. Furthermore, note that the uninvolved vertex is different at each step. Therefore, if we have an even number of vertices, separate one from the rest and apply the above solution to the remaining vertices. At each step, connect the "uninvolved" vertex to the separated vertex. Once all the steps have taken place, each vertex will be connected to the separated vertex, so this is sufficient to insure a proper solution. Therefore, $\frac{\sum i}{\frac{N}{2}}$ steps are necessary to fully connect an even number of vertices. This solution can be trivially turned into a solution to the hand-shaking problem. As noted earlier, this is turn provides a way for N processors to exchange data with each other. An algorithm that follows the strategy outlined here is the method DaSSF-MPI uses for such communications.

12 Appendix B: Machine Specifications

DaSSF 1.22 and DaSSF-MPI for Irix were run on a Silicon Graphics Origin 2000. The operating system present was Irix Release 6.5 IP27. There were 8 R10000 processors present, each with a clock speed of 180 MHz. The system had 1 MB of L2 cache and 6 GB of RAM.

DaSSF-MPI for Linux was run on a network of 4 Gateway 2000 G6-200s running Red Hat Linux 5.2, kernel 2.0.36. Each machine had 2 PentiumPro 200 MHz processors with 512 KB of L2 cache and 128 MB of RAM. The machines were connected with an M2F-SW8 Myrinet switch. Each channel was capable of a 1.28 Gb/s data rate. It should be noted that only one of the processors in each machine was used by DaSSF-MPI during simulation.

The second set of tests, which compared DaSSF 1.22 and DaSSF-MPI using more processors, were run on a Silicon Graphics Origin 2000. The operating system here was also Irix Release 6.5 IP27. There were 16 R10000 processors, each clocked at 250 MHz. 4 MB of L2 cache and 8.5 GB

of RAM were present.

References

[1] R.L. Bargodia and W.T. Liao. Maisie: A language for the design of efficient discrete-event simulations. *IEEE Transactions on Software Engineering*, 20(4):225–238, April 1994.

[2] R.E. Michelsen D.O. Rich. An assessment of the modsim/twos parallel simulation environment. In *Proceedings of the 1991 Winter Simulation Conference*, pages 509–518, 1991.

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

[4] P. Heidelberger and D. Nicol. Conservative parallel simuation of continuous time markov chains using uniformization. *IEEE Trans. on Parallel and Distributed Systems*, 4(8), August 1993.

[5] D. Nicol and P. Heidelberger. Optimistic parallel simuation of continuous time markov chains using uniformization. *Journal of Parallel and Distributed Computing*, 18(4):395–410, Aug 1993.

[6] D. Nicol and P. Heidelberger. Parallel simulation of markovian queueing networks using adaptive uniformization. In *Proceedings of the 1993 SIGMETRICS Conference*, pages 135–145, Santa Clara, CA, May 1993.

[7] D. Nicol and S. Roy. Parallel simulation of timed petri nets. In *Proceedings of the 1991 Winter Simulation Conference*, pages 574–583, Phoenix, Arizona, December 1991.

[8] B.R. Preiss. The yaddes distributed discrete event simulation specification language and execution environments. In *Distributed Simulation 1989*, volume 21, pages 139–144. SCS Simulation Series, March 1989.

[9] Thoman Schmiermund and Steve R. Seidel. A communication model for the intel ipsc/2. Technical Report CS-TR 9002, Dept. of Computer Science, Michigan Tech. Univ, April 1990.

[10] Steve Seidel, Ming-Horng Lee, and Shivi Fotedar. Concurrent bidirectional communication on the intel ipsc/860 and ipsc/2. Technical Report CS-TR 9006, Dept. of Computer Science, Michigan Tech. Univ., November 1990.

[11] J.S. Steinman. SPEEDES: Synchronous parallel environment for emulation and discrete event simulation. In *Advances in Parallel and Distributed Simulation*, volume 23, pages 95–103. SCS Simulation Series, Jan. 1991.