Parallel Import Report

L. V. Kalé, J. Yelon, and T. Knauff
Dept. of Computer Science,
University of Illinois,
Urbana Illinois 61801,
jyelon@cs.uiuc.edu, kale@cs.uiuc.edu

April 17, 1997

Abstract

The following report is a description of the work done on the Parallel Import simulation language developed at the University of Illinois parallel programming laboratory.

1 Introduction

Import [14] is a programming language which was developed at the U.S. Army Corps of Engineering Construction Engineering Research Laboratory. It is a general purpose language with enhancements for discrete event simulation. Its syntax is based on Modsim [5] [3], which in turn was based loosely on Modula-2.

The Parallel Programming Laboratory of the University of Illinois Department of Computer Science received a grant from ARPA to design and implement a parallel version of Import. The objectives of the project were to implement the language runtime system, a compiler, and some demonstration programs. The Parallel Programming Laboratory carried out the project and achieved the stated objectives.

2 Concurrency Extensions for Import

Parallel Import differs slightly from the Import language as described in the US Army CERL's Import Language Reference Manual. This section summarizes the major extensions we have made to the language to enable concurrent processing.

2.1 Object Groups

The primary difficulty in implementing a parallel version of Import lies in the fact that simulation times impose a complete ordering on all method invocations. Such a full ordering is in conflict with

the desired goal of executing many methods in parallel. The imposition of a complete order on methods eliminates all apparent concurrency.

One can regain the concurrency by requiring the user to divide the objects into *object groups*. The members of a group are all located on the same processor (but one may have multiple groups on a processor). Members of a group can interact with each other freely: they can send **tell** invocations to each other, they can invoke each other's **ask** methods, and in general, there is no restriction to how they access each other. However, objects may only interact with objects outside their group via **tell** invocation. Object groups are fundamental to our parallel implementation of Import. To parallelize an Import code, the primary task one must perform is to separate the objects into distinct object groups.

The fact that *object groups* are more or less independent of each other makes it possible for the operating system to process them concurrently. Since object groups only interact via **tell** invocation, the operating system only needs to be concerned with the order it performs the **tell** invocations. Importantly, a **tell** invocation to an object in a group only needs to be ordered relative to other **tell** invocations for the same group, so the operating system can execute **tell** invocations to objects in different groups concurrently.

The user divides objects into groups using a set of extensions to the **new** operator. The four different forms of the **new** statement are:

var := **new** *classname* (*arguments*)

var := **new** *classname* (*arguments*) **remote**

var := **new** *classname* (*arguments*) **on** *processornumber*

var := **new** *classname* (*arguments*) **with** *objectid*

The ordinary form of the **new** operator returns an object in the same group as the creator. This implies that the object is on the same processor as the creator, and that the creator may interact with the newly-created object in any way desired.

The **remote** form of the **new** operator indicates that the system should create a new object-group, and that the new object should be the first member of that group. The group is created on a processor of the system's choosing.

The **on** *processornumber* form of the **new** operator is much like the **remote** form, except that the user selects the processor for the new object and its object-group.

The **with** *objectid* form of the **new** operator indicates that the newly-created object should be inserted into the same group as the specified previously-existing object. The previously-existing object need not be of the same type as the newly-created object — it is used only to identify an object-group.

When choosing which objects to separate into a group by themselves, one must remember the following rules: objects in the same group may interact freely, but the price you pay is that they must all be allocated on the same processor. Conversely, objects in different groups may only

communicate with each other via **tell** invocation, but different groups may be executed concurrently with each other. These limitations/rules must be remembered at all times when programming in parallel Import.

Having stated that objects in different groups can only interact via **tell** method invocation, we now point out a few of the ways that they *cannot* interact:

- It is not possible for an object to communicate with an object in a different group via **ask** invocation. The runtime system keeps track of which objects are in which groups, and will notice attempts to perform **ask** invocations across group boundaries. Such attempts will generate a runtime error. **ask** invocation across group boundaries could be added to the Parallel Import specification, but it would make **ask** invocations slightly more expensive.
- Objects in different groups may not interact by accessing each other's fields. This rule is enforced by the normal semantics of Sequential Import, which does not allow such accesses anyway. However, we felt it was worth mentioning that such accesses can never be a part of the Parallel Import specification, even if they are someday added to the Sequential Import language.
- List nodes can be considered small objects, and thus members of object groups. When a method creates a list node, the list node is in the same group as the object running the method. One cannot traverse a list-pointer to access data belonging to another group. This requirement is enforced silently by copying: when one sends list data in a **tell** invocation, the list nodes are copied, and the copy of the list is a part of the invokee's object group. One must remember that list nodes are copied, because one must dispose the list nodes.

2.2 Global Variables and Global Objects

In any language with concurrency, access to global data must be ordered. The language must provide some sort of restriction that keeps two concurrent threads from randomly interleaving their access to a piece of global data — if the language does not provide such a restriction, chaos will result. In other words, the code must somehow specify groups of accesses that are intended to be executed atomically. We feel that in a language like Import, which is clearly object-oriented, the most appropriate means of defining such atomic critical sections is to use the inherent atomicity of method execution.

The decision we made is to use objects to control access to global data. To implement this restriction, we disallow global variables, except global variables of object types. Global variables of object types cannot be assigned to in Parallel Import, with one exception — they must be *initialized* using a statement of this form:

globalvarname := **new** *classname* (*arguments*) **remote** ;

The compiler notices when a global variable name occurs on the left hand side of an assignment statement, and treats it specially. It checks that the right-hand side is a **new** ... **remote** expression

— assigning anything else to a global variable is a compile-time error. The runtime system verifies that such a statement is executed only once per global variable: the second execution is a runtime error.

Since each global object is in an object group by itself, global objects can be accessed only by **tell** invocation.

3 The Time-Warp Concurrency Control Mechanism

There are two known means of keeping **tell** invocations ordered relative to each other. The first is the Chandy-Misra-Bryant conservative mechanism [7] [6]. The basic idea behind Chandy-Misra-Bryant is that one does not deliver a **tell** invocation to an object in a group until one is certain that the **tell** invocation is the earliest one the group will receive. In other words, the operating system postpones delivery of **tell** invocations until it can prove that no other **tell** invocation with an earlier timestamp will be transmitted.

The second means of concurrency control is the Time Warp mechanism [9], also known as optimistic concurrency control. The Time Warp mechanism works on the following principle. When an object in a group receives a **tell** invocation, the method is executed. If at a later time, another **tell** invocation with an earlier timestamp arrives for an object in the same group, the effects of the incorrectly-ordered method invocation are undone, and the method invocations are redone in the proper order.

The Chandy-Misra-Bryant method has a serious disadvantage: in a language like Import, there is essentially no way to prove that a message with an early timestamp will not be transmitted, except by analyzing the entire system and verifying that no object anywhere is going to send one. In a language like Import, such analyses tend to yield extremely weak results, making it very difficult to execute efficiently. On the other hand, the Time Warp mechanism is not without its disadvantages, the logging overhead being the largest of these. In order to be able to undo the effects of messages, extensive records must be kept. Logging can consume a significant amount of memory (and possibly a significant amount of CPU time as well). The second disadvantage of the Time Warp mechanism is the difficulty of input/output. All input/output must be performed in an undoable fashion. This makes such things as console communication tricky, but not impossible. Given a choice between Chandy-Misra-Bryant and Time Warp, we selected Time Warp, estimating that it would be the less costly of the two.

3.1 Global Virtual Time

As a result of Time Warp's optimistic approach, roll-back logs must be kept such that the objects may roll back their state. In theory, the entire history of a simulation could be kept. However, roll-back logs can be expensive in terms of memory, so in practice the space for the history needs to be reclaimed. Jefferson first noted [9] that at any given point in a Time Warp system, there is a set of unprocessed **tell** invocations, and of those **tell** invocations, one of them has the least timestamp t. There are thus no unprocessed invocations prior to time t. This time t is called the *global virtual time*, or GVT. The processing of a **tell** invocation at simulation time x can alter results computed

at or after simulation time x, but not prior to x. Therefore, no amount of processing will change any results prior to t, the GVT. The GVT thus operates as a sort of event horizon — prior to the GVT, no activity or rollbacks will occur, thus, actions taken prior to the GVT may be committed. Events subsequent to the GVT are speculative and may be rolled back at any time. Thus, if a distributed simulation system can determine the GVT, the roll-back storage for states prior to GVT may be reclaimed.

We have implemented an adaptive GVT algorithm based on message priorities and window boundaries. Each processor element maintains a record of all message activities (sends and receives) occuring in all objects located on that processor element. These events are grouped into *buckets*, where each bucket maintains a record of activities for a particular time slice.

The GVT algorithm performs a reduction of these buckets, grouped into *windows*. A window is composed of one or more *buckets*, determined according to the adaptive algorithm. If the total number of message sends is equal to the number of message receives in a particular time window, the GVT estimation may be potentially advanced. However, it may be possible for two messages, sent from different processor elements at the time of the GVT reduction, to cross each other and cause the GVT algorithm to be misled. Therefore, the GVT estimation is only advanced if two subsequent passes of the reduction give identical results, and the message sends are equal to the message receives.

The algorithm is itself adaptive in two ways. First, all messages for the reduction use Converse's prioritized execution scheme. While prioritized messages do not impose a total global ordering on all messages, they do impose a local ordering per processor element. The algorithm sets the priority of the reduction messages to be the simulation time corresponding to a window several windows ahead of the current GVT estimation. In this way, the GVT reduction message will be delivered at a local simulation time exceeding the next potential GVT estimation. (In fact, the GVT estimation may actually jump multiple windows.) By using these prioritized messages, then, the GVT reduction will stay out of the way of heavy traffic, thus avoiding the overhead of unnecessary GVT reductions.

The algorithm also adapts itself based on its previous results. There are two methods for such self-manipulation: the number of buckets per window, and the number of windows considered in each pass of the reduction. By analyzing previous returns of the reduction, the algorithm can determine if the activitiy within a window was relatively sparse (meaning that the algorithm should take larger "leaps" and thus more buckets per window), or whether there was too much activity per window (meaning that the algorithm should consider fewer buckets per window and thus smaller time slices). Furthermore, if the algorithm has been tending to advance the algorithm just a single window ahead, it can manipulate the number of windows considered at each reduction to either reduce the message-passing overhead, or to advance the estimation by several windows.

3.2 Limitations of the Time Warp Illusion

One of the requirements of Import is that **tell** invocations be executed in the order specified by their timestamps. However, an objective of parallelization is to execute methods concurrently. The Time-Warp protocol pretends to achieve these two contradictory specifications, but of course, Time-Warp is an illusion. Time-Warp, through its undoing and redoing of methods, only pretends

to achieve the correct order of execution. If the illusion is perfect, in other words, if the results produced by the program are identical to what they would have been were the methods executed in the right order, then we are satisfied.

Our Time-Warp mechanism tries very hard to preserve the illusion of correct ordering, and it is almost always successful. In fact, there is exactly one situation in which it can fail. Sometimes, executing methods in the wrong order creates a full crash, a segmentation fault. When this happens, Time-Warp has no opportunity to undo the incorrectly ordered methods.

The consequence to Parallel Import programmers is that they must write programs that won't crash, even if the timestamps are scrambled. If there is a possibility that a method's out-of-order execution could cause a crash, the method must prevent that crash before it happens. It may be necessary to make sure data has been initialized, it may be necessary to check for null pointers or out-of-bounds array accesses: whatever is necessary to keep a segmentation fault from occurring. Remember that a method which is executing out of order does *not* have to compute "correct" results, or in fact, any sort of result at all: it just has to not crash. As long as it doesn't corrupt memory, Time-Warp will recover and redo the methods in the correct order.

One possible modification to the Parallel Import implementation would be to make this checking automatic. It would be quite possible for Import to always check for null-pointers, check array-bounds, and in general, prevent crashing. Such checking would add overhead to the language, and it is usually possible for the programmer to make cheaper checks manually. Nonetheless, automatic checking is worth considering.

4 Implementation Overview

This section gives a very high-level overview of the Parallel Import implementation. Detailed descriptions of specific modules are described elsewhere, the objective of this section is to provide a "big-picture" point of view.

Import has been implemented on top of the *Converse* [10] machine interface. Converse is a toolkit providing subroutines for message transmission, thread creation, scheduling, and many other behaviors common to parallel programs.

All Converse programs (including Parallel Import) utilize the following programming model. Each processor has a task-queue. The life of a Converse program begins when one processor creates an initial task and inserts it into one of the task-queues. Then, all processors begin picking tasks from the task-queues and executing them. The initial task, when executed, creates several more tasks, which in turn create more tasks, and the entire computation proceeds in this fashion. Converse provides messaging subroutines which enable processors to insert tasks into each other's task queues. Therefore, the task-graph branches out across the processors. Eventually, the system runs out of tasks to perform, and the program is done. In Parallel Import, the most important kind of task is the **tell** method invocation.

In Parallel Import, each object is owned by a particular processor. When a **tell** invocation occurs, the processor that owns the object is tasked to execute the **tell** method. This tasking is performed

with the Converse messaging subroutines. Note that the Converse queue is prioritized, and the **tell** invocation task is given a priority equal to its timestamp. Therefore, **tell** invocations with lower timestamps have more urgent priorities.

When a processor extracts a **tell** invocation task from its queue, the system retrieves the object which must perform the **tell** method. A check is then made for out-of-order message delivery: the object's log is checked to see if the object has already executed a **tell** invocation with a higher timestamp. If so, the out-of-order message is undone. Finally, after undoing any incorrectly ordered messages, the system executes the appropriate **tell** method, and the execution of the method is logged.

Executing an import statement is similar to executing a statement in any other programming language. There are a few exceptions, however. The first exception is logging: many statements, before performing their usual action, make an entry in the log recording what they are doing so that they may be undone. The second exception is the few statements that can't be undone: for example, printing a message on the console. Each such statement is treated as a special case. One primary technique for implementing such statements, though, is the "commit" mechanism. Instead of simply printing a message on the console, an entry is made in the *commit-log* recording that a message needs to be printed. The actual printing is delayed until commit-time, which is described below.

Undoing an incorrectly-ordered message generally involves scanning the object's log and undoing each operation. Logging is a complex topic, it is discussed in detail elsewhere. For this overview, we summarize by saying that most statements are simply undone (for example, assignment statements are undone by setting variables back to their old values). The one statement which is more interesting is the **tell** statement: when a **tell** statement is initially executed, a message is sent tasking some processor to execute a method. When the **tell** statement is undone, a second message is transmitted after the first tasking the receiver to cancel or undo the **tell** invocation.

Because of the queueing, these cancel messages may arrive before the **tell** invocation is processed. If so, they are very straightforward: they simply cause the **tell** invocation to be nipped in the bud. However, they may also arrive after the **tell** invocation has been processed. In this event, the cancel-message triggers the undoing of the **tell** invocation. In this way, undos can cascade. We give cancel-messages a very high priority, since it is more efficient to stop a **tell** invocation before it starts than after it finishes.

While all this foreground processing occurs, the GVT algorithm is silently executing in the background. Each time a message is transmitted, the GVT module is notified. Each time one is processed, the GVT is again notified. Therefore, the GVT module can keep count of the number of messages processed for each range of timestamps. It can thus determine the GVT. The details of the GVT module's internal operation are described in a previous section. When the GVT module determines that the GVT has reached a certain time t, the GVT module triggers a cleanup: all undo logs prior to time t are discarded. At this time, the commit-logs are also scanned. Any operation that was entered into the commit-logs prior to time t (and thereby postponed) is finally executed.

5 Estimation of Efficiency

Parallel Import is translated directly into C code. Therefore, most elements of Parallel Import run at the efficiency of C. For example, an addition operation in Import is translated to an addition operation in C. There are only three aspects of Parallel Import that yield potentially slower runtime than an equivalent simulation written sequentially in C. These are:

- Messaging Overheads.
- Logging Costs.
- Rollback Costs.

Parallel Import is much like any other parallel programming language in that one cannot summarize the efficiency of the language in a few sentences or numbers. Instead, we analyze the three different kinds of overhead independently in the following sections. Finally, we present the results of running two test programs to give an overall impression of efficiency in actual Parallel Import programs.

5.1 Messaging Costs

As in any other object-based parallel programming language, the cost of remote method invocation is significantly higher in Parallel Import than the cost of local method invocation. In Parallel Import, **ask** invocation is always local, but **tell** invocation may involve network message transmission if the two object groups are not on the same processor.

The cost of a **tell** invocation includes a number of sources of overhead, which may be summarized as follows:

1. **Packing costs.** The **tell** method's argument data must be copied into a message buffer along with a few bytes of control information.
2. **Transmission costs.** The message must be sent to a remote processor if the target object is remote. If the target object group is local, this cost is bypassed. The cost of this operation varies significantly from machine to machine. We detail these costs in the Converse report. [10]
3. **Queueing costs.** When the message arrives, it must be inserted into a prioritized queue, and eventually, removed from the queue by a scheduler.
4. **Unpacking Costs.** The argument data must be extracted from the message and the **tell** method must be invoked.

Of these, costs 1, 3, and 4 are also being paid by Sequential Import, although the packing and unpacking costs are significantly less for pointer and array data in Sequential Import.

5.2 Logging Techniques

The Time Warp methodology uses logging to make it possible to undo methods. In keeping with the design principle that object groups should be kept independent of each other whenever possible, we have designed the logging system in such a fashion that each object group uses its own set of logs. Each log is structured as follows: the log records a sequence of **tell** -invocations, each with a timestamp. Each **tell** invocation triggers a method, and the method produces a list of the actions needed to undo the method. Therefore, a sample log might appear as follows:

- Processed **tell** -invocation to object 0x000a0010 at time 3034. How to undo:
  - Change memory at 0x000a001c back to 0x19af304b.
  - Change memory at 0x000a0104 back to 0x00000001.
  - Cancel message 0x000102ab (to object 0x000c2230).
- Processed **tell** -invocation to object 0x000aed30 at time 3039. How to undo:
  - Change memory at 0x000aed60 back to 0x01020030.
- Processed **tell** -invocation to object 0x000a0010 at time 3042. How to undo:
  - Change memory at 0x000a001c back to 0x19af304c.
  - Free memory at 0x000e1ecc (which was allocated during method)

Log for object group 0x000b2210

The entries are in temporal order. If, in the example above, the next method to arrive were to have a timestamp of 3041, then the method at time 3042 would be undone, and then the methods at time 3041 and 3042 would be redone in the correct order.

When undoing a method, one need only undo statements that change the state of the object or the system. Such statements fall into three basic categories:

1. Statements that change the contents of memory,
2. Statements that transmit messages,
3. Special statements.

The strategies we use to log the three basic types of operations are described here.

5.2.1 Undoing Memory Writes

Assignments to memory are inherently very cheap operations, and quite common. Adding even a small amount of overhead (for logging) to such statements can result in a significant change to program performance. Therefore, it is necessary to carefully consider the logging strategy used for undoing assignment-statements.

The traditional means of logging assigments statements is to checkpoint an entire object group before every method execution. This makes it possible to undo all the assigment statements of the method *en-masse* by simply restoring the object group to its previous state. The dissapointing aspect of full checkpointing is that it requires the traversal and storage of an entire object group prior to each method invocation, regardless of how little work the method invocation does.

The other alternative is to checkpoint assignment-statements incrementally. Immediately prior to storing a new value in a variable, the old value is logged. This is the strategy used in Parallel Import. If a method modifies every instance variable in the object group exactly once, the cost of incremental checkpointing is a small constant multiple of the cost of full checkpointing. If the method modifies each instance variable more than once, then incremental checkpointing is more expensive than full checkpointing. However, if the method simply modifies one or two instance variables, then incremental checkpointing can be dramatically less expensive.

The cost of incremental checkpointing can be estimated as follows. When using incremental checkpointing, each assignment statement requires a logging operation. The cost of logging increases the cost of the memory write by approximately a factor of 4 on most CPU architectures. It is hard to estimate the overall impact of this slowdown on the program as a whole. If the processor is memory-bound, in other words, if the memory bus is full, and if we use traditional estimates that 90 percent of all memory accesses are reads, then the overall slowdown for memory logging evaluates to 30 percent. In other words, an import program would take approximately 1.3 times as long as the equivalent C program.

Incremental checkpointing and full checkpointing aren't mutually exclusive: it is possible to let some methods perform full checkpointing before they begin, while others perform incremental checkpointing while they execute. The Parallel Import compiler does not yet support this strategy. Supporting it would require the compiler to generate duplicate code: each function would be generated both with and without incremental checkpointing code.

5.2.2 Not Undoing Local-Variable Writes

While assigments to instance variables must be incrementally checkpointed, it is worth pointing out that most assigments are to local variables, and do not need to be checkpointed.

To see the reason behind this, one need only realize that the local variables didn't exist before the method execution, and they no longer exist after the method execution. Therefore, the local-variable state before the method and the local-variable state after the method are automatically identical. There is thus no need to record the changes to local variable state.

5.2.3 Undoing Message Transmissions

Recall that each object group has its own set of logs. When an object group sends a **tell** invocation to another group, logging is performed as follows. The message is given a unique id number. An entry is made in the logs of the sender indicating that a **tell** invocation was transmitted, and indicating the message number and receiver of the message. No other logging information is kept at the sender side. When the message arrives at the receiver's object group, it becomes one of the **tell** invocations in the receiving group's logs.

If for some reason, a **tell** invocation gets unexecuted, a cancel-message is transmitted from the sending group to the receiving group indicating the identification of the message to be cancelled. If the receiver has already executed the **tell** invocation, it will unexecute it. The receiver will undo the specified **tell** method, will discard the message that triggered it, and will then resume processing of other **tell** invocations.

Note that this process can cause cascading rollbacks. It is possible for a **tell** -method to executes a **tell** -invocation, which in turn triggers a **tell** -method, and so forth. Cancellation of the first method will trigger a cancellation of the second method, which will trigger a cancellation of the third, and so forth. This process can be fairly inefficient. Worse yet, it can generate significant message traffic, slowing down even the processors that aren't involved.

To minimize cascading rollbacks, we use two major strategies. The first is prioritization. Cancellations are executed with very high priority: each processor will make sure that cancellations are executed and forwarded as quickly as possible.

The second means of controlling cascading rollbacks is to use a time-window-based throttling mechanism. Processors make a concerted effort to work on messages with similar timestamps. If one processor gets too far ahead of the others, it will stop executing **tell** invocations altogether.

5.2.4 Undoing Special Statements

Some side-effecting statements fall neither into the category of memory writes nor into the category of message transmissions. These statements must be dealt with on a case-by-case basis. Since it would be infeasible for us to describe the undo-strategies for every statement here, we choose a representative example: memory allocation.

To undo a **malloc** operation is quite simple: you free the memory that was allocated. Therefore, whenever a Parallel Import method allocates memory, it simply makes an entry in the undo-logs indicating that when an undo occurs, a free should be performed, and indicating the address of the memory to be freed.

However, undoing a **free** operation isn't possible if the Import runtime has actually called the Unix **free** function. Therefore, when an Import program disposes memory, the memory is not actually freed. Instead, an entry is made in the logs saying *when we become sure that this method won't be undone,* free the following block of memory. This type of entry is called a *commit* entry.

Commit entries are executed when the *global virtual time* advances beyond the timestamp of the

method in question. Global virtual time is defined to be the timestamp of the earliest unprocessed **tell** invocation in the entire machine. Since a **tell** invocation can only trigger activity at or beyond its own timestamp, the global virtual time indicates the time before which no further activity will occur. The computation of the global virtual time is described in a previous section.

5.3 Rollback Costs

Perhaps the biggest overhead in some Parallel Import programs is the occasional need to undo a method invocation. The cost of a single rollback is easy to predict. The amount of time it takes to undo a method is tiny, one simply traverses a table of the assignments that were made and undoes them. Other operations may also need to be undone, but the cost of such undoing is usually negligible. Thus, the cost of a rollback is essentially the cost of unnecessarily performing the work in the first place. In other words, the cost of a rollback is really the cost of a useless method invocation.

Useless method invocations can be broken into two classes: those which Time Warp could have prevented, and those which it could not have. When a processor genuinely has no useful work to do, nothing Time Warp does will be productive. In such cases, *any* activity Time Warp performs will be useless. It should be observed that Time Warp always prefers to execute *something*, where a standard object-based program would idle waiting for a message, a Time Warp program performs useless method invocations. Any problem that would cause poor processor utilization (idling) in a normal object-based parallel language will cause useless method invocations in Parallel Import. Useless work can thus be caused by poor load-balance, by high message latency, and by any of the innumerable other factors that plague parallel object-based programs. Such useless invocations cannot be blamed on Time Warp, since the processor had no real work to do.

However, there are times when a processor performs useless work when it actually had some good work it could have performed. Such avoidable useless invocations can be classified as genuine failures. We can therefore say that the overhead of Time Warp rollbacks is equal to the sum of the execution costs of its avoidable useless invocations. Our system minimizes the frequency of avoidable useless invocations by using the heuristic that the work with the least timestamp is the least likely to be rolled back. This heuristic is implemented by prioritizing messages according to their timestamps.

Unfortunately, it is very hard to predict how many avoidable and unavoidable useless invocations will occur, and in fact, the answer is complex. Thare are innumerable problems that can lead to the situation where a processor has no useful work to perform, and thus skips ahead to useless work and unavoidable rollbacks. There are many other situations that can cause "bad" work with a low timestamp to show up, thereby causing avoidable rollbacks. To sum it up, Parallel Import programs are still programs, they generate almost any behavior imaginable, and thus they can stress the Time Warp system in innumerable ways. Empirical results gathered by other researchers indicate that many programs run quite well (with very few rollbacks), whereas others tend to slow down significantly because of the rollbacks. It is possible to construct programs specifically to cause Time Warp to thrash. We refer the reader to previous research in this area [2] [13] [4] [8] [11] [12] [16]. While it is impossible to make general assertions about how many rollbacks will occur in a Parallel Import program, it is possible to present case studies. We present just such a case-study below.

5.4 The Traffic Simulation

We wished to know how many rollbacks would occur in a "typical" Import simulation. We therefore designed a simulation model which was simple and yet challenging for the Import runtime system. When an object is receiving messages from many different processors, and when those processors are not in sync with each other, it becomes highly likely that an object will receive out-of-order timestamps. We therefore designed this test-simulation to have very high interconnectivity and little locality to stress-test the rollback mechanisms, and to stress-test our software's ability to tolerate out-of-order message reception through the prioritization heuristic.

The traffic model is as follows: the world being simulated consists of a city, which is a 10 x 10 matrix of city streets. At random intervals, cars emerge from parking lots throughout the city. Each car then travels in a more or less direct path toward a destination, where it then vanishes into a parking lot. The frequency of emergence of cars from parking lots is modeled as a simple stochastic distribution. Each car's destination is a randomly-selected intersection, with equal probability assigned to all intersections. The car's path is selected in advance, using a simple directed-travel algorithm with a small random component. The amount of time it takes to traverse each street is modeled as a per-street constant. The amount of time a car spends waiting in line at an intersection is however long it takes the previous cars to get out of the way. Finally, the amount of time needed to actually cross an intersection is a flat simulation-wide constant. The total time it takes a car to reach its destination is thus equal to the sum of the amount of time it takes to traverse the streets, pass through the intersections, and wait in line at intersections. Of these, only the waiting in line at intersections depends upon the behavior of the other cars.

Since all the interaction between cars occurs at intersections and involves waiting for the intersection to be clear, we have implemented the simulation using one object per intersection. Cars are not active objects, they are simply small pieces of data passed about from intersection to intersection. The intersections objects themselves regulate the flow of cars as they pass through the intersections.

We then ran the simulation using two different mappings of intersections to processors. In one mapping, the intersections were scattered randomly across the processors. Therefore, almost every object was receiving messages from four other processors, and was therefore very likely to receive messages out of order. In the second arrangement, we placed the objects in a striped arrangement, with 1 x 5 sections of the mesh assigned to a different processors. This reduces the likelihood that each object will receive out-of-order messages and increases locality, thereby somewhat reducing the number of cross-processor transmissions. We used the same pseudorandom values in each run, therefore, each simulation did the same amount of real work. We used two different parallel processors: the first, an IBM SP1, the second, a network of HP workstations. We also varied the number of processors used. The following two tables show the runtime (in seconds) and the number of methods executed and subsequently rolled back (as a percentage of total methods executed).

runtime

	Random	Striped
SP1 1	13.5 sec	16.2 sec
SP1 2	7.2 sec	7.6 sec
SP1 4	4.1 sec	4.2 sec
SP1 8	3.5 sec	2.5 sec
SP1 16	2.1 sec	1.5 sec
NET 1	33.2 sec	32.3 sec
NET 2	30.6 sec	19.2 sec
NET 4	23.4 sec	16.1 sec
NET 8	13.7 sec	8.3 sec
NET 16	7.8 sec	4.5 sec

rollbacks

	Random	Striped
SP1 1	0%	0%
SP1 2	1%	2%
SP1 4	2%	3%
SP1 8	25%	10%
SP1 16	21%	8%
NET 1	0%	0%
NET 2	5%	1%
NET 4	20%	5%
NET 8	24%	8%
NET 16	6%	2%

As expected, the best speedups were gained on the SP1, using the striped decomposition. The speedup curve on this machine is almost perfectly linear, with a very slight leveling at 16 processors. We conjecture that perhaps a bit more speedup could be gained by going to 32 processors. It is essentially impossible to get good speedups on 64 processors, since the simulation only involves 100 objects. In other words, the speedups we are obtaining are close to optimal for a simulation of this size.

The network of workstations fared almost as well. The speedup curve from 1 to 2 to 4 four processors is not great, which is easy to explain: Ethernet is very slow, there is thus a high cost associated with the decision to send data through the ethernet. On a single processor, none of the data is traveling through a the ethernet. On two processors, perhaps half the data is traveling through the ethernet. By the time one reaches four processors, almost all the data is going through the ethernet. Thereafter, the cost of using the ethernet has already been paid, and speedups become close to linear. Clearly, the Time Warp mechanism is successful in masking the high network latency.

The random decomposition fared significantly worse than the striped decomposition, as expected: the lack of locality results in more communication. However, it should be observed that the random decomposition obtained surprisingly good speedups on the SP1, which can perhaps be accounted for by the SP1's relatively inexpensive communication mechanisms. This suggests that even random decomposition is acceptable on machines with low communication costs, as long as there is sufficient parallelism to allow Time Warp to mask the latency.

The number of rollbacks, overall, was quite low. Even in the worst cases, the amount of wasted work never exceeded 25% of the total work done. With the striped decomposition, the amount of wasted work never exceeded 10%. There was an overall tendency for the number of rollbacks to increase as the number of processors increased. This tendency is caused by the thinning out of the workload: as processors have less and less to do, the probability of their doing useless work increases. Finally, we note that the number of rollbacks dropped off at 16 processors, a minor anomaly which we cannot currently explain.

6 State of the System

This section of the paper discusses the compiler and runtime system, as it stands, and as we expect it to progress.

6.1 Evaluation of Machine Independence

One of the primary requirements of the implementation was that it be, insofar as possible, machine independent. This objective has been met.

To help achieve this objective, we relied upon the Converse parallel runtime system [10], developed at the Parallel Programming Laboratory. The objective of Converse is to support the implementation of parallel languages in such a way that they are both portable and interoperable. Converse currently runs on the following computers:

- The Intel Paragon
- The Thinking Machines CM-5
- The NCUBE-2
- The IBM SP-1
- Networks of Workstations

This is only the current list. Converse and its predecessors have been supported on a wide variety of parallel machines in the past: for example, the Encore Multimax, the Sequent Symmetry, the Intel ipsc/860, and so forth. New versions are slated for the Cray T3D, the SGI PowerChallenge, and the Convex Exemplar. New versions of Converse are created regularly as parallel machines are invented, and old versions are phased out as machines become obsolete.

Parallel Import will therefore run on all of these machines. No source language changes are needed to port a parallel Import program. Some parameters may need adjustment, however. In particular, it may be necessary to group objects into larger object groups when running on machines with high latency, to avoid some of the message transmission costs.

There is only one other anticipated source of difficulty: the Time Warp protocol accumulates log information, which is cleaned up occasionally by a communication-intensive algorithm called GVT. The faster GVT runs, the less memory the logs take. Therefore, on machines with very slow message transmission (networks of workstations), the logs may tend to get large. We have not run simulations where this has been a problem, but then again, we do not have the resources to implement large, real-world simulations. On machines with reasonably low latency, memory use should not be a problem.

6.2 Features Omitted for Technical Reasons

Some constructs were omitted or altered in parallel import for technical reasons. Of these, some were incompatible with the implementation of concurrency. Others were left out because of problems in the language specification as it stands. A list follows:

- **Overloading of non-methods.** We allow overloading of functions and methods, but not overloading of types and variables. Our conferences with the CERL representative indicate that this feature is error-prone, and is likely to be removed from the language, therefore, we chose not to allow it.
- **Synonym Declarations.** According to the language specification, the semantics of synonym declarations are ambiguous in cases of overloading. We advocate the use of less ambiguous constructs like the type-definition operator.
- **The block statement.** Our conferences with the CERL representative indicate that this construct is rarely used, and that some consideration was being given to removing it from the language.
- **Redefining the new operator.** Given the constraints of the time-warp rollback system, it is impractical for the user to implement his own memory management schemes in parallel import. We therefore disallow the redefinition of **new** .
- **Non-Object Global Variables.** Because of the concurrency in the system, it is necessary to define atomic access to global data. Therefore, we require that all global variables be of object-types.

6.3 Features Omitted because of Resource Limitations

As a result of partial funding and thus human resource limitations, we predicted that it would be necessary to reduce the scope of the project from its original objectives. We resolved to make such reductions with the following two ideals in mind:

- **Demonstration Potential**: The system should present a fully-operation demonstration of the feasibility of a parallel Import. Features were to be omitted only if they are orthogonal to the problem of parallel simulation, and if they are not needed to validate the idea of parallel Import.
- **Extensibility**: The implementation should be designed in such a way that it is at possible to extend it to include support for the full Import specification.

As it turns out, it was not necessary to make as many reductions as were anticipated: the system is far closer to complete than we had expected, given the magnitude of the project. Nonetheless, Import being a large language, there were a number of constructs that had to be left out. The following is a list of the constructs that were omitted as a result of human resource limitations:

- **The Dome query language.** The Dome query language is a programming language in itself, and therefore adding it would require a significant investment of effort.
- **The Source-Code database.** The original design for Import included a Smalltalk like source-code database. This is clearly a major undertaking, and we felt that it was orthogonal to our objectives. Currently, we have a single-file compiler. Since the code database is not present, flashback declarations are not available in the language.
- **The Standard Library.** Only a tiny fragment of the standard library has been implemented. Adding functions is fairly straightforward, though nontrivial, because of Time Warp. In particular, I/O is tricky, if it has to be made backtrackable.
- **A number of minor constructs.** The import language contains a great volume of simple constructs. While individually, these are easy to implement, they collectively represent a great proportion of the development cost. Examples of such constructs include the string-concatenation operator, prefix and postfix comments, boolean logical operators like exclusive-or, backslashes in strings, identifier literals, and automatic type promotion.
- **Inheritance, the Tagged type, and Lists.** This includes the **asa** and **isa** operators. Since inheritance and tagging are essentially variants of each other, they should probably be implemented together. Collectively, these are clearly a significant part of the Import language (especially lists), but they will require significant effort.

7 Compatibility Between Parallel and Sequential Import

The US Army Construction Engineering Research Laboratory created a sequential implementation of the Import Language [14]. The compiler for Sequential Import is currently at a reasonably functional level.

One of our major objectives was to make Sequential and Parallel Import compatible with each other. We felt that it should be possible for a simulation programmer to write code that works under both Sequential and Parallel Import. Therefore, while we have had to make some modifications to the language, they were all designed in such a way that the modifications could be easily added to the Sequential compiler as well, keeping the two languages "in sync" with each other. The following list summarizes the extensions/modifications that should be made to the sequential compiler if it is desired that the two languages be kept consistent with each other.

Modification 1. We added object-group handling to the **new** operator.

var := **new** *classname* (*arguments*) **remote**

var := **new** *classname* (*arguments*) **on** *processornumber*

var := **new** *classname* (*arguments*) **with** *objectid*

The sequential Import compiler could be made to accept this construct by parsing and otherwise ignoring the three object-group clauses.

Modification 2. We added a delta-time clause to the **tell** statement and the **dispose** statement.

tell *objectid* **to** *methodname* (*arguments*) **in** *delta*

dispose *objectid* **in** *delta*

Where *delta* is an expression evaluating to an integer. The timestamp of the **tell** invocation or dispose operation is equal to the simulation time at which it was transmitted plus the delta. The **in** clause may be omitted, in which case a delta-time of zero is used. We feel that this is a simple, yet valuable extension both to the parallel and the sequential language, and we recommend simply adding it to the Sequential compiler.

With those two modifications, it should be possible to write code that works equally well under the Parallel and Sequential import compilers.

It is worth noting that a program written in Parallel Import and compiled with the Parallel Import compiler can be executed on a uniprocessor workstation. Therefore, it is possible to do without a sequential compiler. It has yet to be seen whether there is any significant advantage to using the sequential compiler over the parallel compiler in terms of efficiency. Although it would be easy to modify the sequential compiler to accept Parallel Import programs, there is nonetheless some question about whether it is worthwhile to do so, given that one can execute Parallel Import programs on a uniprocessor anyway.

8 Software Installation

This section describes the procedure for installing the Parallel Import compiler and runtime system, and the steps for compiling and running Parallel Import programs.

Parallel Import is based upon the Converse parallel programming system, and parts of it are implemented in the Charm++ programming language. Therefore, before installing Parallel Import, one must download and install the combined Charm/Charm++/Converse distribution. The instructions for doing so are provided in the document entitled *The Charm/Charm++/Converse Installation and Usage Manual*, which is available from the Parallel Programming Laboratory world-wide web page at *http://charm.cs.uiuc.edu/*.

Once a version of Charm and Converse have been installed, you must make sure that the charm compiler **charmc** is in your PATH. Having done so, you can begin to install Parallel Import. Obtain the Parallel Import source code as a tar-file (it is available from the web-page as *http://charm.cs.uiuc.edu/distrib/import.tar.Z*). Uncompress and untar the file, thereby creating a directory named *import*. Change directory into the import directory:

```
% uncompress import.tar.Z
% tar xf import.tar
% cd import
% ls
```

```
SUPER_INSTALL  compiler   pgms    report    runtime
```

The subdirectory **compiler** contains the source code for the Import compiler, **runtime** contains the runtime system, **pgms** contains the demo-programs, and **report** contains the LaTeX source code for this report. The script **SUPER_INSTALL** will build the import compiler, runtime, and demo programs. It takes the following arguments: a word specifying what to install (usually *all*), the name of the version of charm to use (eg *sp1*, *cm5*, *paragon-sunmos*, etc), and command line options for the compiler (eg, *-g*, *-O*). A typical install-command (for networks of solaris workstations with optimization enabled) would be:

```
% SUPER_INSTALL all net-sol -O
```

The sample installation command would produce the following directories:

import/net-sol/bin

import/net-sol/lib

import/net-sol/include

import/net-sol/pgms

import/net-sol/tmp

The *tmp* directory is used only during the installation process, it may be deleted afterward. The *pgms* directory contains the demo programs, in source and binary form. The *bin, lib, and include* directories contain the files necessary to compile Parallel Import programs.

Compiling a Parallel Import program is a simple process: one first converts the program into C using the Parallel Import translator *icompile*. The C code must then be compiled, doing so requires the use of the header files in the Parallel Import *include* directory. Finally, the object file thereby created must be linked with the import runtime system *libimport.o*. The easiest way to perform these actions is with the help of Converse's compilation script **charmc** . For example, if you were compiling a program named *myprog.id* for networks of solaris workstations, you would use these commands:

```
% import/net-sol/icompile myprog.id
% charmc -c myprog.c -Iimport/net-sol/include
% charmc -o myprog myprog.o import/net-sol/lib/libimport.o -language charm++
```

Note that one may bypass the use of *charmc*, but one must then manually specify the Charm/Converse include directories and runtime libraries.

The process required to execute the import program varies from machine to machine. Refer to *The Charm/Charm++/Converse Installation and Usage Manual* for instructions.

References

[1] M. Abrams and P. F. Reynolds Jr., editors. *6th Workshop on Parallel and Distributed Simulation (PADS92). Proceedings of the 1992 SCS Western Simulation Multiconference on Parallel and Distributed Simulation, 20-22 January 1992, Newport Beach, California*, volume 24, no. 3 of *Simulation Series*. SCS, 1992.

[2] D. Ball and S. Hoyt. The Adaptive Time-Warp Concurrency Control Algorithm. In *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 174–177, 1990.

[3] R. F. Belanger. MODSIM II - A Modular, Object-Oriented Language. In *Proceedings of the Winter Simulation Conference*, pages 118–122, 1990.

[4] O. Berry. *Performance evaluation of the Time Warp distributed simulation mechanism.* PhD thesis, University of Southern California, 1986.

[5] O. F. Bryan Jr. MODSIM II – An Object Oriented Simulation Language for Sequential and Parallel Procesors. In *Proceedings of the 1989 Winter Simulation Conference*, pages 172–177, 1989.

[6] R. Bryant. Simulation on a Distributed System. In *Proceedings of the 1st International Conference on Distributed Computer Systems*, pages 544–552, 1979.

[7] K. Chandy and J. Misra. A non-trivial example of concurrent processing: Distributed simulation. In *Proceedings of COMPSAC*, pages 822–826, 1978.

[8] A. Gupta, I. F. Akyildiz, and R. Fujimoto. Performance Analysis of Time Warp with Homogenous Processors and Exponential Task Times. In *ACM SIGMETRICS, Performance Evaluation Review*, pages 101–110, 1991.

[9] D. Jefferson and H. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism. In *Proceedings of the Conference on Distributed Simulation*, pages 63–69, July 1985.

[10] L.V. Kalé, M. Bhandarkar, N. Jagathesan, S. Krishnan, and J. Yelon. Converse: An Interoperable Framework for Parallel Programming. In *International Parallel Processing Symposium 1996 (to appear)*, 1996.

[11] J. P. Kearns and J. E. Payne. Performance Evaluation of Protocols for Distributed Systems: An Alternative to Message Counting. In *Proceedings of the Winter Simulation Conference*, pages 441–445, 1990.

[12] R. J. Lipton and D. W. Mizell. Time Warp vs. Chandy-Misra: A worstcase comparison. In [15] *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 137–143, 1990.

[13] V. Madisetti, D. Hardaker, and R. Fujimoto. The MIMDIX Operating System for Parallel Simulation. In [1] *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92)*, pages 65–74, 1992.

[14] Vance P. Morrison. Import/dome language reference manual. Technical report, US. Army Corps of Engineering Research Laboratory, ASSET group., 1995.

[15] D. Nicol, editor. *Distributed Simulation, Proceedings of the SCS Multiconference on Distributed Simulation, 17-19 January, 1990, San Diego, California.*, volume 22, no. 1 of *Simulation Series.* SCS, 1990.

[16] S. Turner and M. XU. Performance Evaluation of the Bounded Time Warp Algorithm. In [1] *Proceedings of the 6th Workshop on Parallel and Distributed Simulation (PADS92)*, pages 117–126, 1992.

A The Traffic Simulation Code

The following is the code for the traffic simulation. Note that this simulation uses a lot of pseudo-random numbers: the lengths of streets, the origin locations of cars, the car destinations, and the travel paths are all randomly selected. We wished these random numbers to be deterministic, for testing purposes. We therefore used external C subroutines for the random number functions, so that we could feed in the sequences of numbers we desired. Hence the many C functions declared at the top of the file. The braces are comments, they are frequently used to comment out debugging code.

```
module main

    cfunction print();
    cfunction simtime():integer;
    cfunction currentPE():integer;

    cfunction delay_time():integer;
    cfunction dest_x():integer;
    cfunction dest_y():integer;
    cfunction car_dir():integer;
    cfunction street_length():integer;

    import intersection;
    intersectime := 3;
    lastlaunch := 2000;
    xsize := 10;
    ysize := 10;
    grid = array [12] of array [12] of intersection;


    intersection = object

        nextdep    : integer;
        xpos, ypos : integer;
        neighbours : array [4] of intersection;
        roadlength : array [4] of integer;
        spawncount : integer;
        linked     : boolean;

        export constructor method intersection(in x,y:integer)
            i : integer;
        begin
            if (x <> 0) then
                linked := false;
                nextdep := 0;
                spawncount := 0;
                xpos := x;
```

```
        ypos := y;
        for i := 0 to 3 do
            roadlength[i] := street_length(xpos,ypos,i);
        end for;
        tell self to launch() in delay_time(xpos,ypos,0) + 1;
    end if;
end method;

export tell method link(in n0,n1,n2,n3:intersection)
begin
    linked := true;
    neighbours[0] := n0;
    neighbours[1] := n1;
    neighbours[2] := n2;
    neighbours[3] := n3;
end method;

tell method launch()
    carnum : integer;
begin
    if (linked = false) then return; end if;
    if (simtime() <= lastlaunch) then
        carnum := spawncount;
        spawncount := spawncount + 1;
        tell self to sendcar(carnum,xpos,ypos,0);
        { print(xpos,",",ypos,",",simtime(),
            ": launched ",carnum,",",xpos,",",ypos); }
        tell self to launch() in
                        delay_time(xpos,ypos,spawncount);
    else
        { print(xpos,",",ypos,": launches complete."); }
    end if;
end method;

tell method arrival(in carnum,xorig,yorig,age:integer)
    now, deptime, xdest, ydest : integer;
begin
    if (linked = false) then return; end if;
    xdest := dest_x(xorig, yorig, carnum);
    ydest := dest_y(xorig, yorig, carnum);
    if ((xdest=xpos) and (ydest=ypos)) then
        { print(xpos,",",ypos,",",simtime(),": ",
            carnum,",",xorig,",",yorig," done."); }
    else
        now := simtime();
        deptime := now + intersectime;
        if (deptime < nextdep) then deptime := nextdep; end if;
        nextdep := deptime + intersectime;
```

```
            { print(xpos,",",ypos,",",simtime(),": ",
                carnum,",",xorig,",",yorig," here."); }
            tell self to sendcar(carnum,xorig,yorig,age)
                                                in (deptime-now);
        end if;
    end method;

    tell method sendcar(in carnum, xorig, yorig, age : integer)
        direction : integer;
    begin
        if (linked = false) then return; end if;
        direction := car_dir(xorig, yorig, carnum, age, xpos, ypos);
        tell neighbours[direction] to
            arrival(carnum, xorig, yorig, age+1)
            in roadlength[direction];
    end method;

end object;

function init_grid(in g:grid)
    x,y : integer;
    n0,n1,n2,n3 : intersection;
begin
    { initialize the sentinels to null }
    for x := 0 to 11 do
        g[x][0] := null intersection;
        g[0][x] := null intersection;
        g[x][11] := null intersection;
        g[11][x] := null intersection;
    end for;
    { create the real intersection objects }
    for y := 1 to 10 do
        for x := 1 to 10 do
            g[x][y] := new intersection(x,y) on ((x-1)/5) + (y*2);
        end for;
    end for;
    { link them together }
    for y := 1 to 10 do
        for x := 1 to 10 do
            tell g[x][y] to
                link(g[x+1][y],g[x][y+1],g[x-1][y],g[x][y-1]);
        end for;
    end for;
end function;

main = object
    thegrid : grid;
    export constructor method main()
```

```
        begin
            init_grid(thegrid);
        end method;
    end object;

    main:main;

end module;
```