Proceedings of the 1990 Winter Simulation Conference
Osman Balci, Randall P. Sadowski, Richard E. Nance (eds.)

PARALLEL SIMULATION OF THE SHARKS WORLD PROBLEM[†]

Rajive L. Bagrodia
Wen-Toh Liao

Computer Science Department
University of California at Los Angeles
Los Angeles, California 90024

ABSTRACT

The Sharks World problem has been suggested as a suitable application to evaluate the effectiveness of parallel simulation algorithms. This paper develops a simulation model in Maisie, a C-based simulation language. With minor modifications, a Maisie program may be executed using either sequential or parallel simulation algorithms. The paper presents the results of executing the Maisie model on a multicomputer architecture using the space-time simulation algorithm.

1. INTRODUCTION

Parallel simulation refers to the execution of a simulation program on a parallel (or distributed) architecture. A number of algorithms [Misra 1988; Chandy and Sherman 1988a, 1989b; Reynolds 1982; Jefferson 1985], broadly classified into *conservative* and *optimistic* categories, have been suggested for parallel simulation. Simulation and empirical studies have been devised to evaluate the performance of various algorithms on both shared memory and message-passing architectures [Reed et al. 1987; Fujimoto 1988a, b; Su and Seitz 1989; Jefferson et al. 1987; Chandy and Sherman 1989a]. The primary goals of the studies have been to determine whether (variations of) some algorithm may be used to minimize simulation overheads for a specific architecture and/or a set of applications. A number of simulation languages have also been developed to facilitate the design of parallel simulations. Most languages were designed to work with a specific parallel simulation algorithm. Examples include RISE [Marti 1988], ModSim [West and Mullarney, 1988] and TWOS [Jefferson et al. 1987] which only use optimistic algorithms and an Ada-based environment [Sheppard et al. 1988] which uses a conservative algorithm. A few languages like Maisie, were designed to be algorithm-independent, in that the simulation program may be executed using a variety of algorithms. With minor modifications, Maisie programs may be executed using either a sequential simulation algorithm or a parallel algorithm based on *null* messages [Misra 1986], conditional events [Chandy and Sherman 1989a] or optimistic space-time computation [Chandy and Sherman 1989b].

This paper has two goals: develop a Maisie model of the sharks world problem and present the results on the execution of the model on sequential and parallel architectures. The sequential implementation of Maisie used a single event-list and a global simulation clock. The parallel implementation was executed on a multicomputer architecture using the space-time simulation algorithm [Chandy and Sherman 1989b]. The Maisie program for both implementations was essentially identical. The next section describes some important features of the Maisie simulation language. Section 3 gives a brief description of our implementation of the space-time simulation algorithm and also discusses parallel execution of Maisie programs. Section 4 describes the sharks world problem and the Maisie model. Section 5 presents the experimental results of executing different configurations of the model on sequential and parallel architectures. Section 6 is the conclusion.

† This research was partially supported by a grant from NSF (CCR 88 10376)

2. MAISIE SIMULATION LANGUAGE

Maisie [Bagrodia and Liao 1990] is a distributed simulation language derived from May [Bagrodia et al. 1987]. The central construct in the language is the notion of an entity. A Maisie entity-type models physical objects (or a collection of objects) of a given type. An entity-instance, henceforth referred to simply as an entity, models a specific object in the physical system. Interactions among the physical objects are modeled by message exchanges among the corresponding entities.

An entity is created dynamically on a specific processor and cannot be migrated subsequently. Communication among the entities is based on buffered message-passing. Every entity has a unique message-buffer. A message is deposited in the message-buffer of an entity on the execution of an *invoke* statement. Each message carries a timestamp, which corresponds to the simulation time at which the corresponding invoke statement was executed. Messages sent by one entity to another are delivered to the destination buffer in FIFO order.

An entity accepts messages from its message-buffer by executing a *wait* statement. The wait statement has two components: an integer value called wait-time (t_c) and a Maisie statement called a resume block – a (non-empty) sequence of resume statements. A resume statement is like a guarded command, where the guard consists of a message-type (say m_t) and an optional boolean expression(say b_i). A resume statement is said to be *enabled* if the message-buffer contains a message of type m_t, which if delivered to the entity would cause b_i to evaluate to *true*; the corresponding message is called an *enabling* message. If the buffer contains one or more enabling message, in the most commonly used form of the wait statement, the message with the earliest timestamp is removed from the buffer and delivered to the entity. If two enabling messages have the same timestamp, they are processed in an arbitrary order. The only exception is in the case of a *timeout* message (this special message-type is described below) which is delivered to an entity only if the buffer does not contain any other enabling message. By selecting the guards appropriately, the wait statement may be used to ensure that an entity accepts a message from its input buffer only when it is ready to process the message.

On the execution of a wait statement, if the buffer does not contain any enabling messages, the entity is suspended for a *maximum* duration equal to its wait-time t_c; if omitted, t_c is set to an arbitrarily large value. If no enabling message is received in the interval t_c, the entity is sent a special message called a *timeout* message. An entity must accept a timeout message that is sent to it. A non-blocking form of receive may be implemented by specifying t_c=0.

If a wait statement contains exactly one resume statement and its guard specifies timeout as the message-type, the entity will resume execution only when it receives a timeout message after the wait-time specified in the statement has elapsed. As this timeout message cannot be canceled, it is referred to as an unconditional timeout message. Wait statements that schedule an unconditional timeout message are used frequently and are often abbreviated by a *hold* statement. The example at the end of the section illustrates their use in a simulation. If the wait statement contains multiple resume statements, only one of whose guards include timeout as the message-type, the entity may resume execution on the receipt of a message other than timeout. Thus, the timeout message scheduled by such statements is referred to as a conditional timeout message.

As a simple example, consider the simulation of a preemptible priority server in Maisie. In the physical system, the server receives two types of requests, respectively referred to as *high* and *low*, where the requests of the first type have a higher priority and can interrupt the server if it is currently serving a request of type *low*. Figure 1 and figure 2 describe the Maisie model of the system. In the interest of brevity, the program ignores issues concerned with simulation initiation and termination.

```
entity server{cmeanh,cmeanl}
 int cmeanh,cmeanl;
{  int hcnt = 0, lcnt = 0, remtim,lostart;
   message high;
   message low;
   while (true)
      wait until
      { mtyp(high) : {hold(exp(cmeanh)); hcnt++;}
        | mtyp(low) :
         {  lostart= clock(); remtim= exp(cmeanl);
            while (remtim>0)
              wait remtim until
              {  mtyp(high) : {
                      remtim-=clock()-lostart;
                      hold(exp(cmeanh)); hcnt++;}
               | mtyp(timeout) : {remtim=0; lcnt++;
              }
         }
      }
}
```

Figure 1. Maisie Model of Priority Server

Entity-type *server* models the priority server and *hisrc* and *losrc* respectively model the sources for the two types of requests. The *server* entity defines two types of messages, *high* and *low*, to represent the two types of requests that may be received by it. Henceforth, we will use *high* message to mean a message of type *high*; similarly for *low*. The body of the entity consists of a **wait** statement in an infinite loop. When idle, the entity accepts the next message from the buffer by executing the wait statement. If the buffer is empty, it waits indefinitely for the next message. If the entity accepts a *high* message, it executes a hold statement to schedule an unconditional timeout message for the future time at which the service of the request will be completed. On receiving the timeout message, it simply increments the count of requests that have been serviced and reexecutes the wait statement. If the entity services a *low* message, it schedules a conditional **timeout** message; this message will be automatically rescheduled, if the entity receives a *high* message in the interim. In this case, the *high* message is again processed by executing a hold statement, after which the service of the *low* message is resumed. The service time for a message is sampled from an exponential distribution.

```
entity hisrc{srvrid,meanh}
 e_name srvrid; int meanh;
{  while (true)
      /* delay entity by inter-arrival time */
   {  hold(exp(meanh));
      invoke srvrid with high;
   }
}
entity losrc{srvrid,meanl}
 e_name srvrid; int meanl;
{  while (true)
   {  hold(exp(meanl));
      invoke srvrid with low;
   }
}
```

Figure 2. Maisie Model of High and Low Source

The two source entities simply generate appropriate requests at periodic intervals sampled from an exponential distribution. The hold statement in each source is used to delay the entity by the appropriate time-interval; after the time has expired, the entity sends the appropriate request message to the server.

The next section discusses parallel execution of this program. In particular, it considers the execution of this program using an optimistic simulation algorithm.

3. PARALLEL EXECUTION OF MAISIE

In principle, a Maisie program may be executed using either conservative or optimistic algorithms. This paper studies the performance of Maisie programs when executed using the space-time simulation algorithm [Chandy and Sherman 1989b]. In this section, we briefly describe the space-time algorithm and our current implementation.

In the space-time paradigm, multiple logical processes (lp) may be used to simultaneously compute the state of a physical process at different points in time. We use the terms lp and entity interchangeably. Let T be the upper bound on the time for which the system is to be modeled. Let $p^{x,y}$ refer to the lp responsible for the simulation of some physical process in the interval $[t_x, t_y)$, $t_x < t_y$; exactly one lp computes the behavior of a physical process for every t in [0,T]. A *precedence* relation, symbolized by $\rightsquigarrow$, is defined between two lp, where $p^{x,y} \rightsquigarrow q^{x_1,y_1}$ if and only if the state of q^{x_1,y_1} depends on the state of $p^{x,y}$ or on some message received from $p^{x,y}$. If $p^{x,y} \rightsquigarrow q^{x_1,y_1}$, we say that $p^{x,y}$ is a *predecessor* of q^{x_1,y_1} and q^{x_1,y_1} is a *successor* of $p^{x,y}$. Note that although the exact predecessor or successor set for an lp cannot be determined a priori, a loose upper bound on these sets can typically be determined (a trivial bound is the entire set of lp in the system).

Given that the preceding set of lp is executed on a distributed architecture, the correct state of each lp is computed by using the following iterative strategy: given some state for its predecessor lp, an lp computes an *estimate* of its final state. During this computation, it generates a (possibly empty) sequence of messages for each of its successors. The message sequence is sent to each successor after a process has computed its final *estimated* state. When a process gets a message sequence from one of its predecessors that is different from the one it received in its previous iteration, the process recomputes its behavior. This procedure is repeated until eventually the computation reaches a fixed-point where further execution of any process does not change its state, and the computation is said to have converged. A complete description of the algorithm and sufficient conditions for the convergence of the computation may be found in [Chandy and Sherman 1989b].

We now consider parallel execution of Maisie using this algorithm. In a Maisie program, each entity is created on a specific processor. Entities mapped to the same processor are executed sequentially and those mapped to different processors are executed using the space-time algorithm. To implement this algorithm, the run-time system must perform the following major tasks:

- checkpointing and recomputation of an entity.
- creation of multiple *incarnations* of an entity to compute the state of the corresponding physical process at different points in time.
- synchronization of entity incarnations.
- convergence detection to determine the time upto which the simulation has been computed correctly.

As algorithm-independence is a primary goal of Maisie, each of the preceding tasks was implemented transparently by the run-time system. Different strategies may be used to implement the preceding tasks. For instance, the state of an entity may be checkpointed after every event or may be done less frequently. Convergence detection may be done synchronously or using an asynchronous algorithm. Each of these decisions impacts the execution efficiency of a Maisie program. In this section, we describe the current implementation. Alternative strategies are discussed in section 5.

The run-time system checkpoints an entity after processing any message other than a timeout message. Recomputation is initiated whenever the timestamp on a message delivered to an entity's message-buffer is smaller than the timestamp on the last message processed by the entity. As explained in [Bagrodia and Liao 1990], this recomputation strategy reduces rollbacks for some applications.

In the current implementation, at most two incarnations exist simultaneously for every entity. Let $p^{x,y}$ and $p^{y,z}$ be the two incarnations that respectively execute the entity over two successive time-periods, $[t_x,t_y)$ and $[t_y,t_z)$. In a given iteration, both $p^{x,y}$ and $p^{y,z}$ are executed one after the other; however, the output message sequence generated by the execution of $p^{x,y}$ is transmitted before execution of $p^{y,z}$ is initiated. The two incarnations are executed repeatedly until $p^{x,y}$ converges, at which point, another incarnation of the entity is created. Let LP_a refer to an incarnation of some entity. Convergence detection is merged with message communication among the incarnations as follows. Let

- S_a^i be the sequence of messages generated by LP_a in its i^{th} iteration.
- R_a^i be the sequence of messages received by LP_a after executing its i^{th} iteration.

On receiving R_a^i from all its predecessors, LP_a executes its $(i+1)^{th}$ iteration. It then sends the suffix of S_a^{i+1} that is different from S_a^i together with the time t_c upto which consecutive sequences were identical, to each of its successors. The timestamp t_c is simultaneously broadcast to other processes in the system and is used by every lp to compute the time upto which the entire simulation has converged (also referred to as GVT in the Time-Warp system). Note that each of the preceding tasks is transparent to the programmer.

Consider parallel execution of the priority server simulation described in the preceding section. Assume that the *server*, *losrc* and *hisrc* entities are created on different processors. Figure 3 demonstrates entity creation. The *new* statement is used to create a new instance of an entity-type; the *at* clause is used to specify the processor on which the entity is created. If omitted, the entity will be created on the same node as the creator entity.

```
entity main {};
{e_name h1,l1,s1;
   s1=new server{10,10} at proc_a;
   h1=new hisrc{s1,10} at proc_b;
   l1=new losrc{s1,10} at proc_c;
      ⋮
}
```

Figure 3. Entity Creation

For the source entities, the run-time system can transparently detect that the entities do not receive any messages other than timeout. This implies that these entities can never be rolled back, and their states need not be saved (this optimization is not currently implemented in the system). The optimistic execution of the *server* entity is interesting. Unlike other optimistic systems, the *server* entity does not generate a service completion message when it starts to process a *low* message. Rather it schedules a conditional timeout message, which will be generated, only if the entity buffer does not have a *high* message with an earlier timestamp. This ensures that optimistic execution is initiated at a processor only when no messages remain to be processed.

4. SHARKS WORLD PROBLEM

The physical system consists of a 2-dimensional toroidal ocean that contains two types of creatures: sharks and fish. Each creature moves with a constant velocity. The lifetime of a fish is simple: it continually swims in the ocean until it gets eaten by some shark. A shark's life is slightly more interesting: whenever it is within a specific radius a of some fish, it eats the fish. Initially, the sharks and fish are assumed to be uniformly distributed in the ocean. We will use n_f and n_s to respectively refer to the initial number of fish and sharks in the ocean.

The model makes the following simplifying assumptions:

- All creatures are point objects
- Collisions between like creatures have no effect
- Eating is instantaneous

- If a fish may simultaneously be eaten by two or more sharks, any (and only one) shark (chosen non-deterministically) eats the fish.

If the creature velocities are assumed to be static, the simulation program is straightforward: initially, for each fish, compute the earliest time at which it will be within the attacking range of some shark. As the creature velocities remain constant, the attack times cannot change. The simulation may then be executed by simply stepping through the kill events in their correct chronological order. However, in a more realistic model, the shark and fish velocities may change dynamically. In this case, for every change in the velocity of say a shark, the attack time for all the fish in the system must be recomputed. If the total number of creatures in the system is large, this operation is expensive. One way to reduce the computation is to divide the ocean into a number of sectors and have each sector compute its earliest kill event. A change in the velocity of a shark will then affect only the fish that lie within the sector. Division of the ocean into sectors also permits the simulation to be executed on parallel architectures, where each sector may potentially be executed on a separate node. Multiple sectors, however, introduce additional events involving the entry and exit of creatures among the sectors as also the problem of handling fish that may simultaneously be visible to sharks in multiple sectors. We describe how the model with multiple sectors is programmed in Maisie.

4.1 Maisie Model

In developing the Maisie model, the aim was to design an efficient program *that is not biased towards a specific simulation algorithm.* Other things remaining the same, the efficiency of a Maisie program is determined by the number of events generated in the system and the amount of computation performed for every event. As described subsequently in this section, the Maisie model does not generate duplicate events or include extraneous computation for a particular event.

Each sector in the ocean is modeled as a separate Maisie entity. Creatures are represented by appropriate data structures within the corresponding sector. Each sector entity contains two lists, *slist* and *flist* which contain the respective creatures that are visible (as defined subsequently) to the sector. The movement of a creature from one sector to a neighboring one is modeled by appropriate message transmissions between the corresponding sector entities.

Due to the non-zero attack range of a shark, a fish in one sector may be killed by a shark in a neighboring sector. To minimize the synchronization that would otherwise be necessary between neighboring sectors, we extend the boundaries of each sector by a in all directions. The region within the extended boundary will be referred to as the *sphere of influence* or *soi* of the sector. This creates regions of the ocean that lie within the *soi* of multiple sectors with any fish in the region being simultaneously visible to the corresponding sectors. However, at any time, the fish may physically be in exactly one sector, which is referred to as the owner. Figure 4 shows a region with four sectors in which the western *soi* line of sector S2 and northern *soi* line of S4 have been marked. In the figure, fish f1 is visible only to sector S1, f2 to sectors S1 and S2 and f3 to all four sectors. All three fish are owned by S1. Note that a shark is always visible to and owned by, exactly one sector. A fish is in the *flist* of a sector, if and only if it is within the sector's *soi*. A shark is in the *slist* if and only if it lies within the sector boundaries. The major tasks processed

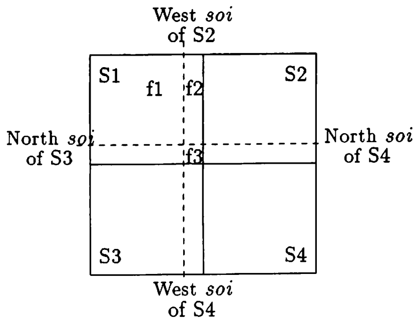

Figure 4. Sphere of Influence

by each sector are as follows:

- death of a fish that is within the *soi* of the sector.
- arrival and exit of a shark to and from the sector.
- arrival and exit of a fish to and from the *soi* of a sector.

We first consider creature arrivals. When a shark enters a sector, the program computes t_x, the time at which the shark will exit the sector, and t_k, the earliest time at which some fish from the sector's *flist* will be within the attacking range of the shark. If $t_k \leq t_x$, both the kill event and exit event are scheduled for the shark; otherwise only the exit event is scheduled. Similarly, when a fish enters the *soi* of a sector, the simulation computes t_k, t_x and t_s, where t_k is the earliest time at which the fish will be within the attacking range of some shark from the *slist*, t_x is the time at which the fish will exit the sector's *soi*, and t_s refers to the time at which the fish will cross the *soi* boundary of some neighbor that it was not visible to earlier (a fish may cross upto two such boundaries). If $t_x > t_k$, an exit event is scheduled for the fish, otherwise the kill event is scheduled. In addition, the appropriate number of *soi* crossing events, henceforth referred to as *visibility* events, are scheduled. Note that at most one future event can be scheduled for each Maisie entity. The preceding events for each creature are scheduled in an internal queue, the earliest among which is then scheduled as the next event for the entity.

For each creature arrival, t_k, t_x and t_s may be computed by using simple coordinate geometry. Given the current position and velocity of a shark and a fish, a quadratic equation can be set up and solved to determine the time at which the fish will be within attacking range of the shark. To compute t_x, the time taken by the creature to cross both the horizontal and vertical sector (or *soi*) boundary is computed and t_x is set to the minimum of the two times. Additionally, the velocity of the creature is used to determine the neighboring sector to which the creature will exit. We now consider how each of the three events, kill, exit and visibility are handled in the simulation.

First consider the kill event. Assume that a shark in sector s_i kills some fish f_i. If f_i is not within the *soi* of any other sector, s_i simply removes f_i from its *flist* and updates the statistics. If f_i is in the *soi* of multiple sectors, the kill event must be broadcast to other sectors. This is accomplished as follows: sector s_i removes the fish from its *flist* and sends a *kill* message to the sector s_o that owns the fish. (Recall that every fish has a unique owner sector.) On receiving this message, s_o removes f_i from its *flist*, and sends a *delete* message to all sectors, other than s_i, that have f_i within their *soi*. If a sector receives a *delete* or *kill* message and the corresponding fish is in its *flist*, it is removed; otherwise the message is ignored. Note that in a parallel implementation, two or more sectors may simultaneously kill a given fish. We prove that the preceding implementation guarantees that eventually each fish is 'killed' at most once in the simulation.

Assume sectors s_i and s_j simultaneously kill a fish f_i at times t_i and t_j respectively. Assume that f_i is owned by s_o at time t_i and by $s_o^{'}$ at time t_j. When processing the kill event, both s_i and s_j will remove f_i from their *flist* and send a *kill* message for f_i respectively to s_o and $s_o^{'}$. Assume $t_i<t_j$. On receipt of the *kill* message from s_i, s_o will remove f_i from its *flist*, and send a *delete* message timestamped t_i to every sector that contains f_i in its *flist*. At t_i, f_i must be in the *soi* of s_j. If not, the visibility event for s_j was scheduled erroneously and will eventually be canceled, causing the kill event for f_i scheduled by s_j to also be cancelled. If f_i is in the *soi* of s_j at t_i, s_j must eventually receive the *delete* message from s_o. Receipt of this message will cause s_j to be reexecuted transparently. In its recomputation, s_j will process the *delete* message at time t_i. As a result it will remove f_i from its *flist* and unschedule its own kill event for the fish. The preceding argument also holds if $t_j<t_i$.

Assume $t_i=t_j$; it follows that $s_o=s_o^{'}$. Without loss of generality, assume s_o first receives the *kill* message form s_i. Once again s_o will send a *delete* message to s_j and subsequently discard any *kill* message it receives from s_j. As $t_i=t_j$, on receiving the *delete* message s_j will not be rolled back. In any case, as it had removed f_i from its *flist* when processing its own kill event, it simply discards the *delete* message. Since each kill is registered only by the owner sector, f_i's kill is registered exactly once by s_o, on its receipt of the first *kill* message.

We show that subsequent rollback of any process results in a consistent eventual state for each sector. Suppose that s_o is rolled back, such that in its reexecution it receives the *kill* message from s_j before that of s_i(recall that messages with the same timestamp may be processed in arbitrary order). Sector s_o will behave exactly as described in the previous paragraph, except that it will send the *delete* message to s_i. Once again, s_i will simply discard the message. Instead of s_o, assume s_j is rolled back and reexecuted such that it receives the *delete* message sent by s_o before it processes its own kill event. In this case, the situation is identical to that described for the case when $t_i<t_j$. Reexecution of s_i can similarly be shown to result in a consistent handling of the kill event for f_i.

We next consider the exit event. When a shark exits a sector, it is removed from its *slist* and an appropriate arrival message is sent to the neighboring sector to which it exits. Similarly, when a fish exits the *soi*, it is removed from the sector's *flist*; however the arrival of a fish in the *soi* of a neighbor must be handled differently from the arrival of a shark.

The simultaneous visibility of a fish to multiple sectors complicates the arrival event for a fish. In general, it is desirable to schedule exactly one visible message for every fish in the *soi* of a sector. A naive implementation may generate a visible message whenever a fish crosses an *soi* or sector boundary of a sector and thus generate multiple visible messages. As each visible message initiates a chain of computation, it is desirable to generate exactly one visible message for the entry of a fish into a sector's *soi*.

In order to avoid duplicate messages, we distinguish between the entry of a fish to a sector and to its *soi*. When a fish enters a sector's *soi*, the program computes only t_k and t_x and schedules one or the other event as discussed earlier. Computation of t_s is done only when and if a fish enters a sector (as opposed to the *soi*). How does a sector determine that a fish will enter its boundary? The sector that sends a visible message simply includes the id of the sector (if any) to which the fish will exit. Alternately, to conserve memory, this may be calculated by a sector on receipt of the visible message. (Note that if a fish enters a sector s_i, s_i must have received a visible message earlier).

When a visibility event occurs, appropriate messages must be sent to the sectors whose *soi* boundary has been crossed. Consider a fish that crosses an *soi* boundary while it is in sector s_j. s_j will send a visible message to a neighbor s_i, only if the following two conditions, respectively referred to as A1 and A2, are satisfied:

- s_i can view the particular boundary that is crossed
- the co-ordinates at which the crossing takes place is *not* already within the *soi* of s_i

We prove that the preceding implementation guarantees a unique visible message for each entry of a fish to a sector's *soi*. We first prove that at least one visible message must be generated for every crossing. Assume that a fish f_i is in the *soi* of a sector s_i, but s_i has not received any visible message for f_i. Consider the earliest *soi* boundary of s_i crossed by f_i. Assume that f_i is in sector s_j when the boundary is crossed. From conditions (A1) and (A2) and the assumption that a visible message was not sent to s_i, it must be that f_i was within the *soi* of s_i. But this violates the assumption that the visibility event was the earliest entry of f_i into the *soi* of s_i. The result follows.

Assume that a sector s_i receives a visible message for a fish f_i from sector s_j and another from s_k. Further assume that the messages are respectively timestamped t_1 and t_2. If $t_1<t_2$, it follows that when f_i crossed an *soi* boundary of s_i in sector s_k, it must already be in its *soi*. Due to condition (A2) s_k will not generate the visible message. We can argue similarly for $t_2<t_1$. Assume $t_1=t_2$. The visible message is sent by a sector only if the fish is in the sector. As a fish may be in at most one sector at any point, $t_1=t_2$ implies that, s_j and s_k must be the same sector. As a sector schedules at most one visible message for every neighbor, we conclude that the assumption is erroneous.

We summarize the preceding discussion by describing the primary message types used by a sector entity:

- *c_enter*{*c_type*,*c_rec*}: message sent by a sector s_i to a neighbor s_j ($s_i \neq s_j$), when a shark leaves s_i to enter s_j or a fish that is in s_i crosses into the *soi* of s_j. *c_type* indicates the creature type, *fish* or *shark* and *c_rec* contains relevant information about the creature (velocity, position, ...).
- *c_exit*{*c_type*,*c_rec*}: A sector sends this message to itself when a shark leaves the sector, or a fish its *soi*.
- *kill*{f_i,s_i}: If sector s_i kills a fish that it does not own, it sends this message to the owner sector.
- *delete*{f_i}: On receiving a *kill* message, the owner sends this message to all sectors, other than the killer sector, that include f_i within their *soi*.

4.2 System Initialization

In executing the sharks world problem with multiple sectors, program initialization is an important concern. The number of events generated in the model depend on the initial configuration of the creatures. As each experiment was executed for a constant simulation time, it is imperative that the system be initialized identically and independently of the number of sectors that are used. One way of guaranteeing identical initial configuration is to store the initial creature configurations in a file and initialize every experiment using that configuration. However, for parallel executions, this imposes some overhead as the data may have to be input sequentially from the front-end.

We adopt a different approach to ensure identical initial configurations independent of the number of sectors used in an experiment. Creature initializations are handled by a special entity called *init*. Let M be the upper bound on the number of sectors. In every simulation exactly M instances of *init* are created. Each *init* entity corresponds to a unique region of the ocean and is assigned a unique serial number. Let c be the number of creatures in the system (for simplicity, assume c is a multiple of M) and N be the actual number of sectors used in the experiment. Each *init* entity is associated with exactly one of the N sector entities. The *init* entity generates a sequence of random numbers (using its serial number as the initial seed) to determine the initial position and velocity of each of its c/M creatures and sends the initial configuration to the corresponding sector entity. For a given number of creatures, each *init* entity must generate the same sequence of random numbers, independent of N. This ensures that an identical initial creature configuration is generated for each simulation that is executed for a given number of creatures.

5. EXPERIMENTAL RESULTS

The experiments reported in this paper were executed for a square toroidal ocean which is 65K units on each side. The attack range of a shark was fixed at 50 units. The creature speeds were chosen uniformly at random from the interval [50,200] and the direction chosen uniformly at random from 0 to 360-ϵ, for a sufficiently small, positive ϵ. Every experiment had an equal number of sharks and fish, with the total number of creatures varying from 64 to 2048. For a given number of creatures, the effect of increasing the number of sectors in the ocean on the execution time of the simulation was recorded. Both sequential and parallel experiments were executed with 1,2,4,8,16 and 32 sectors. Each experiments was executed for 5000 simulation time units.

Both parallel and sequential implementations were executed on a Symult S2010 multicomputer. Each node of the multicomputer uses a Motorola 68020 cpu and has 4MB of main memory. The front-end is also a 68020 based Sun-3 workstation. For the sequential implementation, the Maisie run-time system used a sequential simulation algorithm based on a single event-list and a global clock. The results for the sequential implementation are shown in figure 5. For a given number of creatures, the graph plots the cpu time used to simulate the system as a function of the number of sectors of the ocean (note that the x-axis uses a log scale). As seen from the figure, in general, as the number of sectors is increased for a given creature configuration, the execution time of the simulation initially decreases and then begins to increase. This may be explained as follows: for a given number of sectors, each arrival event in a sector causes the next kill event for the sector to be recomputed. As the number of sectors are increased, the average number of creatures per sector decreases, decreasing the amount of computation required to calculate the next kill event. However, increasing the number of sectors also increases the total number of arrival events in the system. The relative contribution of each factor is responsible for the pattern of change in the elapsed time as a function of the number of sectors in the system. In the Maisie program, the largest percentage improvement seems to occur with 2048 creatures, where the elapsed time decreases from 206 seconds for 1 sector to an optimal value of 44 seconds with 32 sectors, Similar behavior is also reported by Conklin et al. [Conlkin et al. 1990]. However, in their simulations (executed on a single node of the Butterfly), increasing the number of sectors reduced the execution time *by a significantly larger factor.*

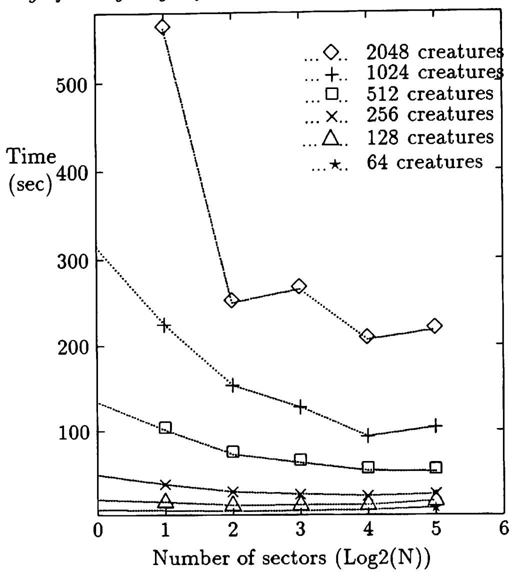

Figure 5. Sequential Execution

This difference may be explained by the different amount of computation required to schedule a kill event in the two programs. In the program described in [Conlkin et al. 1990], each sector entity only records the time for the earliest kill event in the sector. This implies that whenever the sector processes an arrival or a kill event, the kill time for *all creatures in the sector* must be recomputed to determine the next kill event for the sector. If the model has only one sector, this implies an $O(n_f * n_s)$ computation for every arrival or kill event in the system. For a fixed number of creatures in the ocean, as the number of sectors are increased, the number of fish and sharks in each sector decreases by approximately the same factor. However, the complexity of the computation of the next kill event decreases by a *quadratic* factor. As a result, increasing the number of sectors shows an overall improvement in the execution time of the sequential simulation even for configurations with a small number of creatures.

In contrast, in the Maisie program, the earliest attack time for each creature in a sector is calculated when the creature enters the sector and is subsequently maintained in an internal event-list. This implies that when any arrival event occurs, say the arrival of a fish f_i, it is only necessary to compute the attack time for f_i, with respect to all sharks in the sector. The attack time for the remaining fish in the *flist* does not change. Furthermore, after processing a kill event for the sector, the next kill event for the sector entity is selected from the internal event-queue and does not have to be computed all over again. The two factors together imply that other things remaining the same, the average computation per event is significantly less in the Maisie program than in the preceding model. For a fixed number of creatures, as the number of sectors is increased, the number of creatures in each sector again decreases. However, unlike the previous model, the complexity of the computation decreases only by a linear factor.

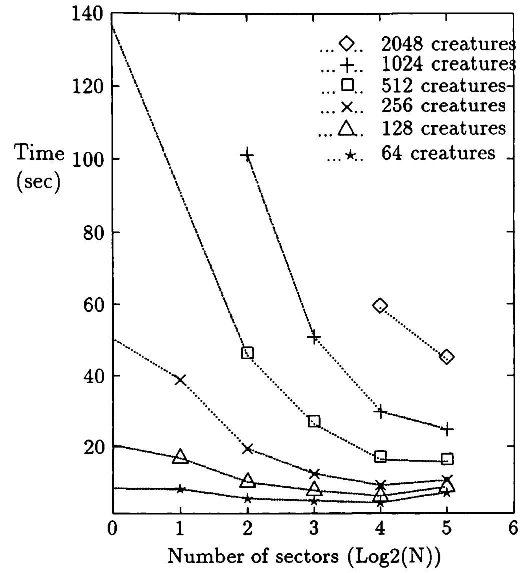

Figure 6. Parallel Execution

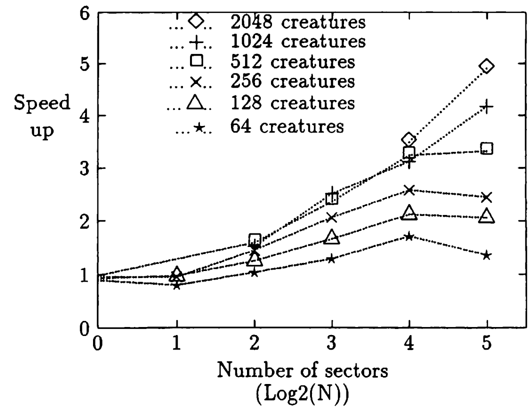

Figure 7. Speedup (t_{par}/t_{seq})

Figure 6 presents the results for the execution of our model on multiple nodes of the Symult. For the parallel implementations, each sector was mapped to a unique node. The figure gives the elapsed time as a function of the number of nodes used to execute the simulation. Once again, the elapsed time decreases as the number of sectors are increased and then begins to increase once again. Note that as the number of creatures is increased, the number of nodes for which the performance of the simulation is optimal, also increases. In general though, for a given number of creatures, the number of sectors for which the sequential implementation was optimal is different than that for which the parallel implementation was optimal. Figure 7 plots the speedup obtained by the parallel implementation, where speedup is measured as the ratio of the execution time of the sequential program to the execution time for the parallel program. As seen from the figure, the speedup increases monotonically as the number of creatures is increased. Figure 8 compares the optimal sequential simulation time with the optimum parallel execution time for each creature configuration studied in these experiments. The number in brackets indicates the number of sectors (which is also equal to the number of nodes for the parallel implementation) that yielded the optimum time for the corresponding configuration. As seen from the figure, the slope of the curve for the parallel implementations is significantly flatter than that of the sequential implementations. This data is replotted in figure 9 as a speedup curve, where the speedup is the ratio of the best parallel time (t_{par}) to the best sequential time (t_{seq}).

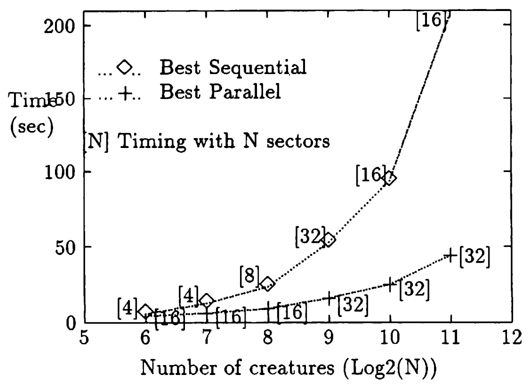

Figure 8. Best t_{par} vs Best t_{seq}

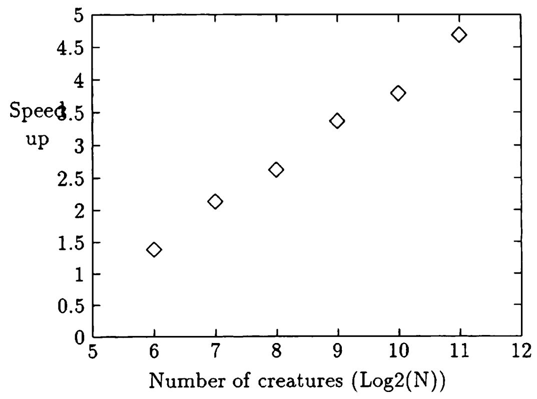

Figure 9. Speedup Over Best Sequential Time

Previous studies on optimistic simulations have indicated that state saving overheads are the primary source of inefficiency for distributed simulations. Many optimistic simulators checkpoint the system after each event (a message is roughly an event). This seems to have the benefit of minimizing the rollback distance, in case of an error. However, if an entity has a large state space and does relatively little computation per event, it would seem more efficient to save state less frequently even at the expense of a slightly longer roll-back. By default, the Maisie runtime system checkpoints the state of an entity after processing any message other than a timeout. We experimented with two other schemes for state saving. In the first variation, the entity is checkpointed after every message including the timeout message. In the second variation, an entity is checkpointed only when it begins to execute every iteration. As seen from figure 10, checkpointing an entity after every event is considerably less efficient then the other two schemes.

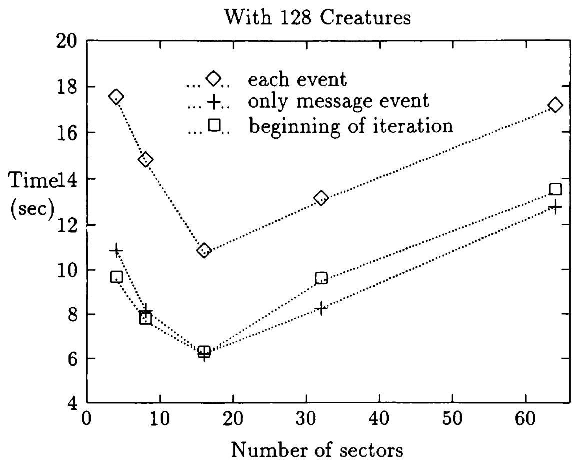

Figure 10. Elapsed Time vs Frequency of Checkpointing

6. CONCLUSION

This paper described the design of a Maisie program to simulate the sharks world problem and presented the timings for both sequential and parallel executions of the model. In writing a Maisie simulation, the programmer is essentially unaware of the specific simulation algorithm that will be used to execute the program. The flexibility of changing the underlying simulation algorithm without having to change the program will be of considerable utility in comparing the performance of a variety of simulation algorithms. For instance, in going from the sequential to the parallel version, the only change that was made to the Maisie program, was the creation of sector (and initialization) entities on different nodes of the multicomputer. This uniformity was of significant benefit in the design stage, as the program could be tested and debugged on a SUN workstation, before being executed on the multicomputer. Program maintenance was also facilitated, as different versions did not have to be maintained for the sequential and parallel implementations.

The experimental results indicate that reasonable speedups may be achieved for a relatively small number of creatures; further the speedups increased linearly as the number of creatures in the system were increased. For a configuration with 2048 creatures, the parallel implementation with 32 nodes and 32 sectors was better by a factor of almost 5 over the best sequential implementation. The performance of the space-time algorithm depends on a number of factors. The more important factors include frequency of checkpointing, time-length for which multiple incarnations of an entity are created and the algorithm used to detect the time upto which the simulation has converged. For the purpose of this study, we experimented with a few variations in the value of these factors. But every variation seemed to suggest many other ways to improve the performance of the algorithm, only a few of which could be tried. Further experience is needed with the system in order to extract optimal performance improvements in the execution of a simulation program with the space-time simulation algorithm.

ACKNOWLEDGMENTS

We are grateful to Mani Chandy for discussions on the space-time simulation algorithm. We are also grateful to Chuck Seitz for arranging unrestricted access to the Symult S2010 at Caltech and to Sharon Brunett for her assistance in using the machine.

REFERENCES

Bagrodia, R.L, K.M. Chandy, and J. Misra (1987), "A Message-Based Approach to Discrete-Event Simulation," *IEEE Transactions on Software Engineering*, SE-13,6,654-665.

Bagrodia, R.L., and W. Liao (1990), "Maisie: A Language And Optimizing Environment for Distributed simulation." In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, D. Nicol, Eds. SCS, San Diego, CA, 205-210.

Conklin, D., J. Cleary, and B. Unger (1990), "The Sharks World: A Study in Distributed Simulation," In *Proceedings of the 1990 SCS Multiconference on Distributed Simulation*, D. Nicol, Eds. SCS, San Diego, CA, 157-160.

Chandy, K.M., and R. Sherman (1989a), "The Conditional Event Approach to Distributed Simulation," In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, B. Unger, and R. Fujimoto, Eds SCS, San Diego, CA, 93-99.

Chandy, K.M., and R. Sherman (1989b), "Space-Time and Simulation," In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, B. Unger, and R. Fujimoto, Eds SCS, San Diego, CA, 53-57.

Fujimoto, R. (1988a), "Lookahead in Parallel Discrete Event Simulation," In *International Conference on Parallel Processing*, D.H. Bailey, Eds Pennsylvania State University Press, University Park, PA, 34-41.

Fujimoto, R. (1988b) "Time Warp on a Shared Memory Multiprocessor," Technical report UUCS-88-021a, Computer Science Dept., University of Utah, Salt Lake City, UT.

Jefferson, D., B. Beckman, and F. Wieland et al. (1987), "Distributed Simulation and the Time Warp Operating System," In *Proc 12th SIGOPS Symposium on Operating Systems Principles*, 77-93.

Jefferson, D. (1985), "Virtual Time," *ACM TOPLAS*, 7,3,404-425.

Marti, J. (1988), "Rise: The Rand Integrated Simulation Environment," In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, B. Unger, and D. Jefferson, Eds SCS, San Diego, CA, 68-72.

Misra, J. (1986), "Distributed Discrete-Event Simulation," *Computing Surveys*, 18,1,39-65.

Reynolds, P. (1982), "A Shared Resource Algorithm for Distributed Systems," In *9th International Symposium on Computer Architecture*, Austin, TX, 259-266.

Reed, D.A., A.D. Malony, and B.D. McCredie (1987), "Parallel Discrete Event Simulation: A Shared Memory Approach," In *Proceedings of the 1987 ACM SIGMETRICS Conference*, ACM, 36-39.

Sheppard, S., C. Davis, and U. Chandra (1988), "Parallel Simulation Environments for Multiprocessor Architectures," In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, B. Unger, and D. Jefferson, Eds SCS, San Diego, CA.

Su, W., and C.L. Seitz (1989), "Variants of the Chandy-Misra-Bryant Distributed Simulation Algorithm," In *Proceedings of the 1989 SCS Multiconference on Distributed Simulation*, B. Unger, and R. Fujimoto, Eds SCS, San Diego, CA, 38-43.

West, J., and A. Mullarney (1988), "Modsim: A Language for Distributed Simulation," In *Proceedings of the 1988 SCS Multiconference on Distributed Simulation*, B. Unger, and D. Jefferson, Eds SCS, San Diego, CA, 155-159.