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

A "CONSERVATIVE" APPROACH TO PARALLELIZING THE SHARKS WORLD SIMULATION

David M. Nicol
Scott E. Riffe

Department of Computer Science
College of William and Mary
Williamsburg, Virginia 23185

ABSTRACT

This paper describes how we parallelized a benchmark problem for parallel simulation, the Shark's World. The solution we describe is conservative, in the sense that no state information is saved, and no "rollbacks" occur. Our approach illustrates both the principal advantage and principal disadvantage of conservative parallel simulation. The advantage is that by exploiting lookahead we find an approach that dramatically improves the serial execution time, and also achieves excellent speedups. The disadvantage is that if the model rules are changed in such a way that the lookahead is destroyed, it is difficult to modify the solution to accommodate the changes.

1. INTRODUCTION

The Shark's World simulation was proposed in early 1990 as a testbed problem for studying issues in parallel simulation [Conklin et al. 1990]. Following that proposal, we were invited to participate in a 1990 Winter Simulation Conference session devoted to different methods for attacking the Shark's World problem. We were asked to write a paper that emphasizes the process by which the problem was parallelized using some sort of conservative synchronization. Our background in parallel simulation has largely been in showing how to extract lookahead (the ability of a simulation model element to predict its future behavior) which can then be exploited by any conservative method. Indeed, our thesis has long been that conservative synchronization protocols ought to be tailored to the specifics of the problem [Nicol and Reynolds 1984].

The Shark's World is a conceptually simple simulation designed to capture many of the salient features of more complex physical models, such as the colliding hockey pucks problem [Hontalas et al. 1989]. The Shark's World has a torodal topology, and is populated with two species: sharks, and fish. A creature moves at a fixed velocity, and a fixed direction; velocity and direction may vary from creature to creature. A shark will eat any fish that strays within a distance A of the shark. The fish disappears from the simulation, but the shark's course remains unaltered.

This problem's principle difficulty lies in the complexity of determining potential interactions. When a fish and shark are relatively close in the domain one may easily enough determine if and when the shark could eat the fish. However, there is no guarantee that the fish will make the rendezvous, as it may be consumed by a different shark at an earlier time. As we will see, the solution proposed in [Conklin et al. 1990] involves a certain amount of event cancelling to retract falsely anticipated interactions.

Lookahead is absolutely essential to achieve good performance using any conservative synchronization method. Our past methods for lookahead computation relied on techniques such as the pre-sampling of random variables [Nicol 1989], and exploitation of non-preemptive queueing disciplines [Nicol 1991]. Identification of lookahead tends to be problem-class specific. When we accepted the challenge to parallelize the Shark's World, we accepted the responsibility to find lookahead in a type of problem we had not yet considered. Indeed, finding that lookahead proved to be the most important aspect of our solution approach.

This paper chronicles our efforts. We began by developing a *baseline* serial simulation along the lines suggested in the orginial Sharks World paper. The purpose of this simulation was to develop a better understanding of the problem, and to provide a benchmark for the eventual parallel simulation. In our implementation all distance and time quantities are taken to be real numbers. This is a minor deviation from the original Sharks World simulation where distance and time are discretized. A discretized approach is at variance with inherently real quantities involved in movement calculations—sines and cosines for example. Next we pondered the simulation problem, looking for exploitable lookahead. Once the lookahead was identified we wrote a new serial simulation which emulates the eventual parallel simulation. The advantage of this intermediate step is that workstations provide a far better development and debugging environment than does almost any parallel system. The new serial simulation employs a different computational paradigm than the original Shark's World simulation, and on a workstation implementation runs over twenty times faster than the baseline simulation. Having thus validated the lookahead ideas we parallelized the new serial code. The parallelization was straightforward—it required only two hours to parallelize, debug, and validate the first parallel version.

This paper is organized as follows. §2 outlines the original sectoring paradigm, and the different approach we adopt. §3 describes our method in more detail, and explains its parallelization. §4 addresses performance, and §5 presents our conclusions.

2. SOLUTION METHODS

Our approach to the problem is different than the one outlined in [Conklin et al. 1990]. As a point of comparison we briefly outline the original simulation strategy, and then our own.

2.1 Original Method

The Shark's World is partitioned into sectors. There are two types of simulation events: `Change_Sector`, and `Attack_Fish`. The former occurs when a fish or shark passes from one sector to another. The latter occurs when a shark attacks a fish. A rough sketch of the basic event processing follows. In the interests of readability, a number of details have been suppressed.

`Change_Sector` Suppose a creature is entering sector c. Determine the identity $\hat{c}$ of the next sector the creature will enter if it manages to pass through c unharmed, and determine the time t_c at which it would leave c. Schedule another `Change_Sector` event for the creature, at time t_c. Finally, call a routine `NewAttackTimes()`. If the entering creature is a fish, this routine computes the minimal next-attack-time (if any) from among all sharks presently able to attack sector c. If the entering creature is a shark the routine computes its next attack time on every fish currently in sector c, possibly re-scheduling an `Attack_Fish` event as a result.

`Attack_Fish` Cancel the event where the fish leaves the sector. Remove the fish from the simulation. Call a routine `NextKillTime()` to reschedule the time of the next shark attack in the sector.

The basic idea behind sectoring is to limit the number of shark-fish interactions that have to be considered in `NextKillTime()`. One chooses (square) sectors that are at least as large in both dimensions as the distance A at which a shark may attack a fish. Then at any given simulation time t, the set of sharks that are able to attack a given fish must reside within one sector's distance of the fish. When computing the time of the next attack in the sector one need consider only the sharks that are close enough to the sector. Alternately, one permits smaller sectors but extends the search for sharks to any sector within distance A.

Computation of the next attack in a sector c has time complexity $O(F_c S_c)$, where F_c is the number of fish presently in the sector and S_c is the number of sharks that can attack fish in c. Therefore, as the sector size decreases the complexity of each `NextKillTime()` call decreases. However, because there are more sectors the total number of such calls increases, and the number of `Change_Sector` events also increases. One must empirically determine the sector size that optimally manages this tradeoff. A complexity analysis given in §4 qualifies this tradeoff.

2.2 Starting Over From Scratch

A conservative solution method must find and exploit lookahead. The basic problem with the Shark's World simulation is that after we schedule a `Change_Sector` event for a fish, the fish may later be consumed by a fast-moving shark whose future presence was unknown at the time we scheduled the `Change_Sector` event for the fish. Where then is the lookahead?

After much deliberation (and a few false starts), we noticed the most obvious of lookahead properties: a shark's position at any future time t can be exactly predicted. For that matter, one can predict the future position of any fish at time t, provided that it is alive at time t. Our first thought was to use the basic sectoring approach, but then continuously "project" shark positions far enough into the future so that whenever a fish enters a sector, all sharks that could possibly attack it during its duration in that sector are already known. We can then accurately compute whether the fish manages to escape the sector, or is eaten (and by whom). If we determine that it escapes we can confidently report its departure to the next sector in its path. Indeed, this is a viable conservative approach to the problem. However, there is a simpler and faster method.

Given the specifications for a simulation, one typically attempts to determine the most efficient way to implement the simulation. When implementing conservative parallel simulation one has to trust that the problem specifics will not change, for within the problem specifics one finds the needed lookahead. In a commercial setting there is a very real danger that mid-way through development a customer will change the problem specifics. This can spell disaster for a conservative approach, for the changes may destroy the lookahead around which the simulation is designed. The Shark's World simulation is an excellent example of this phenomenon.

The object of the Shark's World simulation is to determine the time, position, and cause of each fishes' demise. Now the trajectories of the sharks and fishes are completely determined by their initial positions, directions, and velocities. In theory we can compute the intersection of a fishes' trajectory with a shark's trajectory. By considering all the sharks, we can determine the earliest time at which a shark attacks the fish. The only problem is computing the trajectory intersections. The section to follow will show how this can be efficiently done.

The "back-to-basics" approach has many advantages. We will see that it runs over twenty times faster on a Sun Sparc 1+ workstation than does the sectoring simulation. We will also see that parallelization is trivial, and that excellent speedups are achieved. It is hard to dismiss these advantages. But consider any minor modification to the rules that permit a creature's trajectory to change: as a consequence the lookahead properties are changed, and the entire approach has to be reworked. Herein lies the dual nature of conservative parallel simulation.

3. THE TIP ALGORITHM

The Sharks World problem asks that we determine which fish are consumed within a time interval $[0, T]$, the time, location, and cause of their consumption. If we can efficiently determine the earliest attack time between every fish and shark, the most straightforward way to solve this problem is to compute the minimum attack time (if any) on every fish. We call this *intersection projection*, owing to its implicit projection of creature positions far into the future. We will actually employ intersection projection over different time-slices of the simulation, yielding the name *Time-sliced Intersection Projection*, or simply TIP. This section describes TIP, its underlying method for projecting intersections, and its parallelization.

3.1 Projections and Time-Slices

The intersection projection algorithm can be thought of as a doubly nested loop. Certain efficiencies are achieved if the inner loop runs over sharks, while the outer loop runs over fish. For, within the inner loop, we may maintain the least kill time t_{kill} known so far for the fish fixed as the outer loop variable. Each successive inner loop iteration (i.e., for each successive shark) we need only look for interactions with the fish within the interval $[0, t_{kill}]$—any later interaction will not occur—thereby reducing the workload somewhat. The order in which we compare sharks with a given fish has a great deal to do with the savings we achieve. Consider a fish that is eaten by some shark S_0 early in the interval and would interact (if it had lived) with another shark S_1 late in the interval. If we compute the interaction with S_1 first we project both the shark and fish through most of $[0, T]$ before finding the interaction. If instead we had computed the interaction with S_0 first, we would have been able to cut the projection with S_1 well short of T.

One way to avoid unnecessary projection is to use *time-slices.* Divide $[0, T]$ into subintervals of width Δt. We start by computing all interactions between sharks and fish over $[0, \Delta t]$. Any fish that is consumed in this interval is removed from the fish list. The positions of all remaining creatures are then projected forward to time Δt, and we repeat the process over subinterval $[\Delta t, 2\Delta t]$. We call this *Time-sliced Intersection Projection*, or TIP. TIP has the advantage of limiting unnecessarily long projections, and of reducing the number of fish involved at each subinterval. It does suffer the additional cost of "moving" each creature at the end of a subinterval, and creates the problem of deciding how large Δt ought to be. Informal experimentation with our code showed that approximately a factor of two gain in performance over no time-slicing was achieved using $\Delta t = T/10$. This rule was employed in the experiments reported in §4.

3.2 Intersections in a Torodal World

We wish to determine when a given fish and a given shark are close enough for the shark to consume the fish. The problem is complicated by the fact that both the fish and shark may complete many circuits of the Shark's World before meeting. The solution we present here efficiently deals with this problem.

Let $(x_f(t), y_f(t))$ be the position of the fish at time t, θ_f be its angle of direction and let v_f be its velocity. Similarly define $(x_s(t), y_s(t))$, θ_s, and v_s for the shark. If the fish and shark are to be within distance A, they must be within distance A in each coordinate. Our approach is to determine the functional form of all epochs when the fish and shark coincide in x, and the functional form of epochs when they coincide in y. Around each epoch there is a window within which the fish and shark coordinates differ by no more than A. We look for the intersection of windows around x epochs and windows around y epochs.

For the purposes of description, view the behavior of creatures' x-coordinates, ($x_f(t)$ and $x_s(t)$), as particles on a ring of length M. $x_f(t)$ moves with velocity $v_f \cos\theta_f$, and $x_s(t)$ moves with velocity $v_s \cos\theta_s$; the sign of a velocity indicates the particle's direction (clockwise or counter-clockwise). Without loss of generality assume that the magnitude of $x_f(t)$'s velocity is larger than the magnitude of $x_s(t)$'s velocity. If the two particles are moving in the same direction $x_f(t)$ overtakes $x_s(t)$ at relative velocity $v_{xr} = |v_f \cos\theta_f - v_s \cos\theta_s|$; in other words, after their first meeting $x_f(t)$ and $x_s(t)$ coincide every $P_x = M/v_{xr}$ units of time. If the particles move in opposite directions they approach each other at relative velocity $v_{xr} = |v_f \cos\theta_f| + |v_s \cos\theta_s|$, and meet every $P_x = M/v_{xr}$ units of time. The time lapse T_x until their first meeting is easily determined from the particles' initial positions. Thus, the particles exactly coincide at all epochs

$$t_k = T_x + kP_x \qquad \text{for } k = 0, 1, 2, \ldots.$$

It takes time $I_x = A/v_{xr}$ for the two particles to close from a distance A apart. For every epoch t_k the two particles are within distance A during $[t_k - I_x, t_k + I_x]$. Exactly the same sort of analysis applied to the Y coordinate yields the relative velocity v_{yr}, an initial intercept time T_y, intercept periodicity P_y, and window parameter I_y. Figure 1 illustrates these definitions.

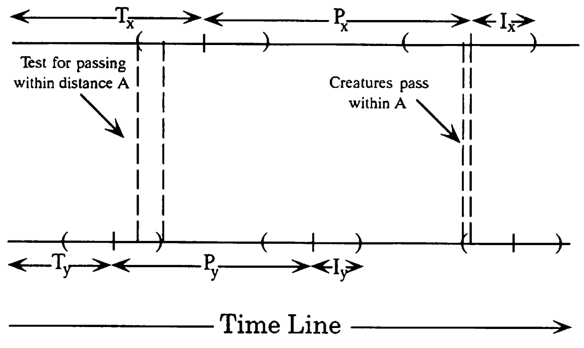

Figure 1. Time line of coordinate projections

A necessary condition for a shark and fish to be within distance A at time t is that t lie in some window around an X-coordinate epoch, and in some window around a Y-coordinate epoch. Let e_x and e_y be the respective x and y epochs, and let $[s_1, s_2]$ be the intersection of the windows around e_x and e_y. At any time $s \in [s_1, s_2]$ the squared distance between the two creatures is

$$D(s)^2 = (v_{xr}s - v_{xr}e_x)^2 + (v_{yr}s - v_{yr}e_y)^2.$$

The time of interest is found by solving for s satisfying $D(s)^2 = A^2$, choosing the least real solution. If no real solution solution exists the creatures do not come within distance A during time $[s_1, s_2]$.

The algorithm for determining the earliest time at which a shark attacks a given fish is straightforward. First one checks to see if the shark and fish are initially placed within distance A. If so the attack occurs immediately. Otherwise we initialize $e_x = T_x$ and $e_y = T_y$. Proceedingly iteratively, we check to see if $[e_x - I_x, e_x + I_x] \cap [e_y - I_y, e_y + I_y] \neq \emptyset$. If the intersection is nonempty we test for an attack; if an attack is discovered we are finished. If the windows do not intersect or intersecting windows fail to produce an attack, we either add P_x to e_x or add P_y to e_x, depending on whether $e_x \le e_y$ or $e_x > e_y$. The process repeats until either an attack is discovered, or the epoch values are larger than the simulation termination time.

In the worst case we will generate all epochs within the simulation time span and not find an attack. Assuming that the maximum creature velocity is bounded from above, the computational complexity of determining the first time of an attack is $O(T)$, where T is the length of the simulation time span. Therefore the overall complexity of determining the earliest attack time on all fish is $O(FST)$, where F is the number of fish and S is the number of sharks.

3.3 Parallelization

The TIP algorithm is very easily parallelized. We simply partition the fish evenly among processors, and ensure that within every time-slice a copy of every shark visits every processor. No communication of sharks is necessary when the problem size is small enough so that every processor may hold a copy of every shark. When there are so many sharks that one processor cannot hold a copy of each we divide the sharks into "groups". A shark group has as many sharks as a single processor can hold. Every processor is given a copy of an entire shark group. If there are k groups and P processors, processors 0 through $P/k - 1$ get group 0, processors P/k through $2P/k - 1$ get group 1, and so on. Each processor computes the interactions of all sharks in its current shark group with all its fish. It then sends the shark group to a processor that has not yet seen a copy of that group. This is accomplished by having each processor j send its current group to processor $(j + P/k)\%P$.

Our implementation on the Intel iPSC/2 permitted as many as 16,382 total creatures to reside on each processor at a time. Models this large are overwhelmingly dominated by the computation cost—hours of execution time can be expected. In the face of this the relative cost of moving sharks around would be trival on problems that require such movement.

4. PERFORMANCE

We consider the performance of TIP in three ways. First, we use a simple performance model to show that while TIP's computational complexity cost per simulation unit time on a fixed domain has order (FS), the complexity of the sectoring approach has order $((FS)^2/N_S + FS\sqrt{N_S})$, where F,S are the numbers of fish and sharks and N_S is the number of sectors. TIP therefore has an algorithmic advantage over sectoring. Secondly, we demonstrate that our approach works faster *serially* than does the sectoring approach. Finally we measure the parallel performance achieved on a sixteen processor Intel iPSC/2 where each processor is based on the 80386/80387 chips, has 4Mb of memory. We analyze performance as a function of problem size, measured by the total number of initially placed creatures and the length T of the simulation time interval. We find that the number of creatures plays the predominant role in determining good performance. Speedups in excess of 8 are achieved when as few as 64 sharks and 64 fish are simulated; speedups quickly approach 15 as the number of creatures is increased.

4.1 Analysis

Complexity results for the sectoring approach can be derived from a simple analytic model. From this model we discover that if the domain is left constant as the number of sharks and fishes increases, TIP has a better asymptotic complexity than does sectoring.

Consider a fixed sized domain where the number of sectors N_S is variable, as are the numbers of fish F, sharks S, and the simulation time interval T. There are three main computational costs.

1. Whenever a kill event is processed, we recalculate the sector's next-kill-time;
2. Whenever a new shark comes within attacking range of a sector we compute its next attack time on every fish presently in the sector;
3. Whenever a new fish enters a sector we calculate the minimum attack time from any shark presently able to attack that sector.

Our performance analysis looks at the costs and frequencies of each of these computations.

For the sake of simplicity assume that all fish and sharks are evenly distributed among the N_S sectors. First we consider the cost and frequency of the next-kill-time calculation. As N_S increases the number of fish in a sector decreases as F/N_S, and the number of sharks decreases as S/N_S. The next-kill-time calculation would seem then to be proportional to FS/N_S^2, however, for large enough N_S the calculation involves more than S/N_S sharks. Any shark within attacking range A of a sector must be considered; the domain within distance A of a sector has an area bounded below by πA^2. The number of sharks involved in a next-kill-time calculation is therefore asymptotically proportional to S, giving the calculation an asymptotic FS/N_S complexity. To analyze the frequency of this computation, view the simulation from a single shark's stationary frame of reference. Imagine a circle of radius A drawn around the shark. Whenever any fish enters that circle it is eaten, and somewhere another next-kill-time calculation occurs. There is a rate λ_A at which a randomly chosen fish crosses into a fixed circle of radius A; ignoring depletion effects the ensemble rate at which any fish enters a given circle is $F\lambda_A$. As there are S sharks, the ensemble rate of kills (and therefore next-kill-time events) is proportional to FS. One can modify this argument to include the effects of depleting fish; however, the end complexities are not altered. Combining the rate (in simulation time) of the next-kill-time calculation and its cost, we see that the computational complexity per unit simulation time is asymptotically proportional to $(FS)^2/N_S$.

The second type of computational cost is suffered whenever a shark comes within attack range of a sector. The perimeter of the attack zone around a sector is at least $2\pi A$ long; therefore the rate at which sharks cross into a given sector's attack zone is asymptotically proportional to S (again a consequence of the domain having fixed size). The calculation is linear in the number of fish in the sector: F/N_S. There are N_S sectors where this calculation occurs. Therefore, the computational complexity per unit simulation time due to this calculation is asymptotically proportional to (FS).

The third type of computational cost is suffered whenever a fish crosses into a sector. One must compute the minimal attack time on that fish from any shark able to attack the sector. This cost is linear in the number of sharks attacking the sector, a number which is proportional to S. The frequency of this computation is the frequency of fish crossing the sector boundary. The length of the sector perimeter is inversely proportional to $\sqrt{N_S}$, so the computation occurs at a given sector at a rate proportional to $F/\sqrt{N_S}$; collectively it occurs in the simulation at rate $F\sqrt{N_S}$. The computational complexity per unit simulation time due to this calculation is therefore asymptotically proportional to $FS\sqrt{N_S}$.

Combining the costs of all three types of computations we see that the overall computational cost per unit simulation time is asymptotically proportional to $((FS)^2/N_S + FS\sqrt{N_S})$. The most efficient sectoring program will adapt the number of sectors to the number of creatures in order to keep the first term low. However, in doing so it increases the second term. The computational cost per unit simulation time of TIP is proportional only to FS.

4.2 Serial Performance

Prior to engaging in any parallelization we sought to determine whether TIP was in fact an efficient solution to the problem (at the time we had not yet done the complexity analysis). The most straightforward means was to compare serial versions of TIP and a sectoring simulation. The results were extremely encouraging. Over a spectrum of problem sizes the TIP algorithm computed simulation behaviour over *twenty times faster* than the sectoring approach. This basic performance differential remained throughout a series of experiments that sought to determine the best sector sizes for the example problems.

There are a whole range of simulation parameters one might vary; given this overly large space of possibilities it seemed to us that varying the parameters most likely to affect performance was a reasonable course of action. The parameters we varied have to do with the size of the simulation: the numbers of creatures, and the length of the simulation interval. All other parameters we left constant, and at the values reported in the original Shark's World paper. These values are given below.

M	65536
A	50
Velocity	Uniformly at random from $[50, 200]$
Initial X	Uniformly at random
Initial Y	Uniformly at random
Direction	Uniformly at random
Simulation Duration	2000 time units

All the measurements we report have equal numbers of fish and sharks initially. We studied problems with total creature populations of 32, 64, 128, 256, 512, 1024, 2048, and 4096. The table below gives the average finishing times for these simulations as implemented on a Sun Sparc 1+ workstation.

Creatures	Sectoring (secs)	TIP (secs)	Sectoring/TIP
32	1.2	0.1	12
64	3.1	0.1	31
128	8.8	0.3	29.3
256	29.2	1.3	22.4
512	107	5	21.4
1024	459	21	21.8
2048	1936	83	23.3
4096	8117	334	24.3

Comparison of Sectoring and TIP on Sun Sparcstation 1+

4.3 Parallel Performance

We studied parallel performance on the same set of problems described above, on a sixteen node Intel iPSC/2 distributed memory multiprocessor. For each parameter setting we executed a set of "short" runs with $T = 2000$ and a set of "long" runs with $T = 100,000$. Our implementation will support simulations with up to 131,072 total creatures. However, the execution times get quite long, so in order to keep the serial exection times within reason we limited speedup computations to runs with rather smaller numbers of creatures. The large problem we have run in parallel required 122 minutes; for this problem $F = 16386$, $S = 16386$, and $T = 2000$.

Figure 2 plots timings taken from the serial version run on one iPSC/2 node (it is interesting to note that there is apparently a factor of five speed differential between a Sparc 1+ and a single iPSC/2 node), and Figure 3 gives the speedups achieved using sixteen processors. Some experimentation suggested that we use a time slice of $\Delta t = T/10$. The value of "speedup" is not as rigorous as we would like: ideally one would exhaustively determine the best time-slice for each serial run and use that in the speedup calculation. In fact, we believe that the cross-over behavior of the short and long speedup functions is likely due to the non-optimality of the $\Delta t = T/10$ rule—in particular, the serial timings for long runs and few creatures are probably inflated owing to this phenomenon. Other caveats include the fact that we did not include initialization time (which matters little because we could have parallelized it had we spent the time on it), nor do we include the IO time required to report the fishes final status.

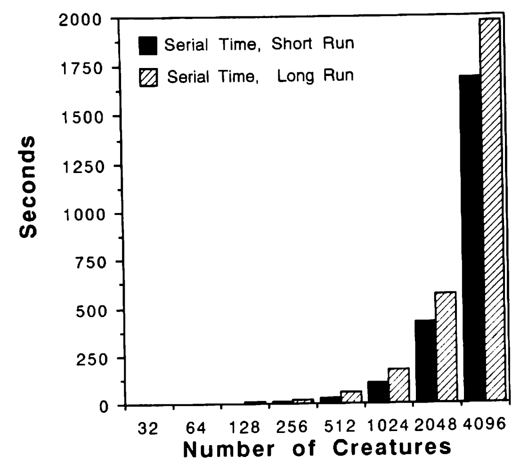

Figure 2. Timings for long and short runs

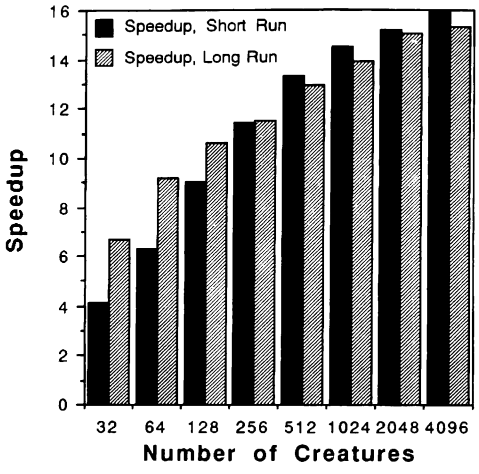

Figure 3. TIP Speedups for long and short runs

The parallelization costs which keep TIP from achieving perfect speedup are due to load imbalance. Our tricks for reducing the number of TIP inner loop iterations for a given fish cause variability in each fishes processing time, as does the fact that the forward projection of a fish and shark can be terminated with the first discovered intersection. Our timings wait for all processors to synchronize globally, thereby waiting for the processor with the heaviest load to complete. However, this degradation will decrease as the number of fish increases, due to central limit theorem effects of reducing the load variance in relationship to the mean.

5. CONCLUSIONS

The parallelization of discrete-event simulations offers many challenges. We examined some of those in the context of a particular model, the Shark's World simulation. We offer two conclusions. First, knowledge and exploitation of lookahead in the simulation model can lead to excellent performance. Our search for lookahead in Sharks World led us to a completely different solution approach. The advantages of the approach are manifold: on a serial workstation problems are solved over twenty times faster than with the "usual" discrete-event approach; the approach is easily parallelized and achieves high speedups. The second conclusion is that excellent performance achieved by exploiting lookahead can be easily thwarted by relatively minor changes in problem specification. Any modification to the model rules that affects lookahead exploitation may require a great deal of modification to the solution approach. This fundamental problem will be suffered by *any* conservative synchronization method whose performance depends on lookahead. To the extent that one can draw general conclusions from this specific example, we conjecture that optimistic synchronization mechanisms may be better suited than conservative methods for a *general* discrete-event simulator; on the other hand, a *specific* simulation may have very good lookahead properties that can be efficiently exploited by a conservative mechanism.

ACKNOWLEDGEMENTS

This work was supported in part by CIT Grant INF-89-014, in part by NASA Grant NAG-1-060, and by the Institute for Computer Applications in Science and Engineering at NASA Langley Research Center.

REFERENCES

Conklin, D., J. Cleary, and B. Unger (1990), "The Sharks World (A Study in Distributed Simulation Design)," In *Distributed Simulation 1990, (Simulation Series 22)*, SCS, San Diego, CA, 157-160.

Hontalas, P., et al. (1989), "Performance of the Colliding Pucks Simulation on the Time Warp Operating System," In *Distributed Simulation 1989, (Simulation Series 21)*, SCS, San Diego, CA, 3-7.

Nicol, D. (1988), "Parallel Discrete-Event Simulation of FCFS Stochastic Queueing Networks," *SIGPLAN Notices 23*, 9, 124-137.

Nicol, D. (1990), "Analysis of Synchronization in Massively Parallel Discrete-Event Simulations," In *Proceedings of the 1990 ACM PPoPP Symposium*, ACM, New York, NY, 89-98.

Nicol, D. and P.F. Reynolds, Jr. (1984), "Problem Oriented Protocol Design," In *Proceedings of the 1984 Winter Simulation Conference*, S. Sheppard, U.W. Pooch, and C.D. Pegden, Eds. IEEE, Piscataway, NJ, 471-474.