Proceedings of the 1987 Winter Simulation Conference
A. Thesen, H. Grant, W. David Kelton (eds.)

Performance Issues for Distributed Battlefield Simulations

*David M. Nicol**
The College of William and Mary
and
Institute for Computer Applications in Science and Engineering

1 Introduction

This paper discusses three performance issues that arose in our implementation of a time-driven battlefield simulation on a medium-scale multiprocessor. The problems we identify are generic, so that the observations we make and the conclusions we draw are applicable to the general class of physical domain simulations which use time-stepping to advance the simulation. The first issue we discuss is that of *mapping* the simulation onto the multiprocessor. Under the message-passing paradigm, the assignment of workload to processors has the single most important influence on performance. The second issue is that of performing redundant computation in order to avoid a certain amount of communication. This issue is important when the cost of communication is high. Finally, we discuss the possibility of deadlock due to distributed contention for message buffers, and outline a solution which insures that deadlock does not occur.

2 A Battlefield Simulation

The model problem for our study is a battlefield simulation based on Zipscreen [2,4], written by John Gilmer of the BDM Corporation. Zipscreen is a simplified version of the CORBAN [3] simulation for the purposes of studying performance issues in mapping battlefield simulations to parallel architectures. Zipscreen and CORBAN represent a battlefield as a two dimensional plane tessellated by hexagons (in addition, CORBAN imposes a hierarchical scheme of hexagons on this domain). Combat *units* move through the domain; units from opposing sides engage in simulated combat when they are geographically close. Figure 1 illustrates the hexagonal plane and combat units.

Both Zipscreen and CORBAN are time-driven, rather than discrete event simulations. There are strong reasons to suspect that the discrete-event paradigm on battlefield simulations will severely limit possible performance gains achievable by parallelism. The problem of avoiding deadlock in distributed discrete-event simulations has been well studied [1,6,14]. A formal treatment in [7] has proven that to avoid deadlock without rolling back simulation clocks, it is *necessary* for certain logical processes to be able to predict their future message-passing behavior far enough into the future to allow some other logical process to advance its clock (deadlock avoidance protocols that rely on prediction demonstrate only the *sufficiency* of behavior prediction). The ability to predict future behavior is very limited in battlefield simulations, implying that the synchronization constraints and overhead of avoiding deadlock are likely to adversely affect performance. The Time Warp [5] mechanism of rolling back clocks avoids the behavior prediction problem, but does so at the cost of extensive memory requirements, and the potential threat of having rollback "thrashing". While Time Warp is an aesthetically pleasing idea, its utility on large real-world problems has yet to be empirically demonstrated. Time-stepped simulations seem to offer the best potential for battlefield simulations, since all computational activity for a time-step can be performed concurrently. However, it is important that the time-step be large enough to allow a significant amount of computation.

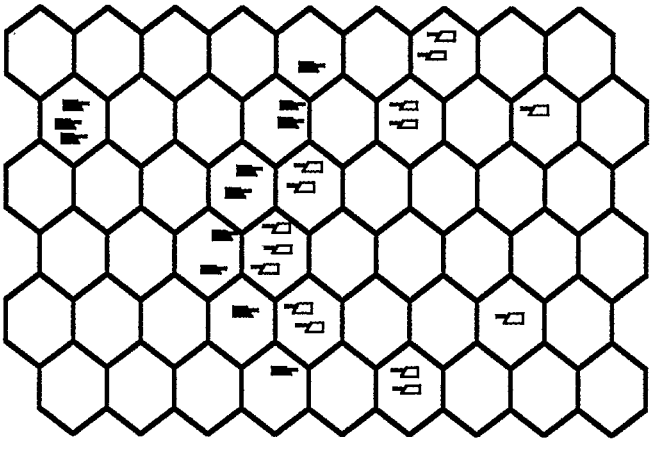

Figure 1: Battlefield Simulation Board

*This research was supported in part by the National Aeronautics and Space Administration under NASA Contract NAS1-18107 while the author was in residence at ICASE, Mail Stop 132C, NASA Langley Research Center, Hampton, VA 23665.

3 The Mapping Problem

Zipscreen focuses on the *perception*, *combat*, and *movement* activities found in CORBAN. At every time-step, a unit perceives others by scanning its resident hex and all directly adjacent hexes for enemy units. The unit then engages in combat with all of these enemy units, and reports the losses it inflicts on those units. Following combat, the unit changes its geographical position, possibly moving to a different hex.

Zipscreen organizes its data by maintaining lists of hexagons on which units reside; for every such hex it maintains a list of resident units. The mapping of units to processors clearly has a significant impact on performance. The efficiency of perception is very much affected by the mapping, since some units will have to perceive units resident on different processors than their own. Combat is computationally intensive, and so the mapping has an important influence on load balancing. Inter-unit (and hence potential inter-processor) reporting of losses is affected by communication costs as well. The efficiency of movement also depends on the cost of communication; the fact that units dynamically move has a profound impact on load balancing and load balancing strategies. Zipscreen's concerns for communication and computational costs in the face of uncertain and changing workload are representative of similar concerns for any simulation of any irregular phenomenon in a physical domain.

Because of the relatively high cost of message passing on current parallel architectures, it is not efficient to employ the type of fully dynamic workload assignment (e.g. idle processors access a central work queue) so effective on shared memory machines. Instead, the workload assignment needs to be semi-static, changing only infrequently. An apparently natural static workload assignment is to simply assign each processor an equal number of units, an approach discussed by Gilmer and Hong, in [4]. Since one unit can conceivably interact with any other unit from an opposing side, this approach requires that every processor communicate (directly or indirectly) with every other processor, if only to say that it has nothing to communicate. In addition to the high communication needs, Gilmer and Hong noted that the approach suffered from load imbalance. This problem arises because serious computation occurs only when units are geographically close, so that at any given time step a unit may or may not demand substantial computation. A simple analytic model derived in [9] demonstrates why significant load imbalance can be expected using their method. The fundamental cause for performance declines due to load imbalance is the extreme sensitivity that globally synchronous algorithms have to load distribution. The execution time of a time-step is determined by the processor having the most to do during that time step. For instance, if the most heavily loaded processor has just 15% of the workload in a simulation using sixteen processors, the *efficiency* (speedup/#processors) is just 41%—on average, a processor is idle 59% of the time simply waiting for the most heavily loaded processor to finish. Because a battlefield simulation's workload is so unpredictable and variable, a static workload mapping is not likely to give high efficiencies for every time step.

Our approach is to map regions of the *domain* to processors, rather than directly assigning units. A processor is responsible for simulating units on its assigned subset of the domain. A two dimensional domain tessellated by hexagons can be viewed as a "rectangular" array of hexagons. This is seen in Figure 2 where the "rows" are clearly defined while the hexes in a "column" zig-zag vertically. For the purposes of partitioning, we assume that the domain consists of a rectangular array of hexes, where each hex can be uniquely identified by its row and column indices. Partitioning consists of covering the domain with rectangles each w hexes wide and h hexes tall (with the possibility of some deviation from these dimensions at the edges of the domain). These rectangles themselves form a rectangular array that we index by "rectangle row" and "rectangle column". We cover the domain by assigning hex (i, j) to rectangle $(i \bmod h, j \bmod w)$. In a similar fashion, we view the N processors as forming a r by c rectangular array of processors. Then, rectangle (k, m) and all the hexes it contains are assigned to processor $(k \bmod r, m \bmod c)$. This scheme is called *wrapping*, and has been studied on a variety of problems [13] . Figure 2 shows a wrapped assignment of blocks with $w = 2$ and $h = 3$ to the four processors P_0, P_1, P_2, P_3 (with the obvious mapping between two dimensional and one dimensional indices).

The communication requirements of this approach are very local. The vast bulk of communication is between processors holding adjacent regions—every processor is logically adjacent to at most six others. Our mapping scheme requires that a processor hold copies of all units in hexes adjacent to those assigned to the processor. The number of these "boundary hexes" (and hence the number of additional units a processor must hold copies of) is strongly dependent on the size (and hence number) of the subregions. Communication is required every time-step between processors to insure that the status of border units is properly maintained. Our strategy of partitioning a domain and exchanging boundary information during run-time is

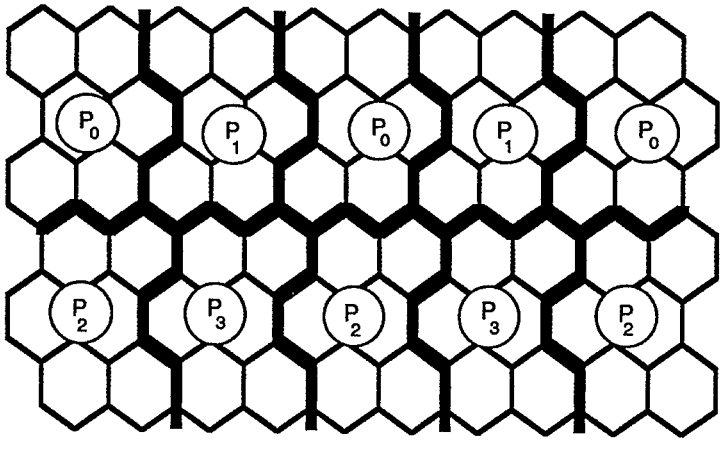

Figure 2: Rectangular Partitioning of Domain

similar to strategies employed by parallel scientific programs [8,15]. In both scientific codes and our battlefield simulation, partitioning the domain into large subregions leads to lower communication costs than does partitioning into small subregions. However, large subregions lead to higher risks of load imbalance. Some degree of load balancing could be achieved simply by assigning adjacent hexes to different processors. This balance comes at the price of increased communication overhead. The values w and h allow a parameterized partitioning of the domain. Smaller values create a finer granularity of workload, and tend to yield a better balance of load. By using a parameterized approach to partitioning, we can control the trade-off between load imbalance and overhead, and find the best granularity for the problem and architecture. The principles underlying this tradeoff are discussed further in [13].

Wrapping exploits the observation that for many problems arising from physical domains, workload tends to be positively correlated in space. Use of wrapping increases the likelihood of breaking up regions of high computational activity for execution by several processors. In a battlefield simulation this is true for two reasons. The first follows from the rule requiring engaged units to lie on the same or adjacent hexes. If there is combat activity on a hex, there is a significant chance that the opposing units lie on different hexes, so that simulated combat occurs on at least two adjacent hexes which might be separated by wrapping. The second reason follows from the observation that battles (and hence battlefield simulations) tend to be localized in space. The knowledge that a particular hex contains an engaged unit makes it likely that the hex lies in a region where the main battle-lines are drawn. Wrapping increases the chance of giving each processor a section of the battle-line.

Our version of Zipscreen currently runs on the Flex/32 Multicomputer at the NASA Langley Research Center. The Flex has twenty processors, two of which serve as hosts; the remaining eighteen are used for parallel processing. Each processor is NS32032 based, and has approximately 1M bytes of local memory. There is a global memory with approximately 2.25M bytes. Zipscreen uses the global memory only to implement message passing between processors. Since the bulk of inter-processor communication costs are related to costs of message handling and not to actual transmission, the Flex/32 implementation should fairly well represent performance achieved by message-passing architectures with fast communication channels but not necessarily fast access protocols.

Our experiments used data sets which simulate a battlefront in 32×32 and 64×64 hex domains. An imaginary line intersecting both the top and bottom rows of the domain is drawn; this line separates units from opposing sides. Each side has 500 units, distributed randomly within a corridor several hexes wide abutting the dividing line. The units' directions are set so that the opposing sides become closer. The simulation is run for fifty time-steps, during which time the two sides largely pass through each other. Early time steps tend to require a heavy amount of battle simulation, while later time steps require substantially less. This mixture of activity is intended to measure the "average" performance of the distribution scheme.

Figure 3 plots the measured performance of a representative run using sixteen processors on a 32×32 domain, as a function of the degree of granularity. The value n of the horizontal axis is the number of hexes arranged in a square) in a logical block. The timings exclude the time required to load the simulation on the processors, but do include I/O required during the run. A typical speedup for this type of problems is 8.5; the average speedup when using eight processors is 5. These speedups are actually quite reasonable, considering the dynamic nature of the simulation and the static nature of the mapping. The performance does leave significant room for improvement; dynamic remapping schemes such as those discussed in [10,11,12] offer promise of even better speedups.

Our implementation of Zipscreen divides a time-step into a computation phase, followed by a communication phase. During the computation phase every processor is engaged in the perception and combat activities. At the beginning of the computation phase every processor has an updated local copy of any unit which may engage with some unit owned by the processor. The computation phase generates damage reports, which are then exchanged among processors during the communication phases, along with reports of movement of units between processors. This structure allows us to measure the processor efficiency during just the computation phase, and hence measure the effects of our mapping scheme on computation costs in near-isolation from its effects on communication costs. Figure 4 plots average processor efficiency during the computation phase as a function of the size of the (square) hex blocks assigned to processors. This data was taken from a run using sixteen processors on a 64×64 hex domain. It is clear that our intuition behind the mapping of domain to processors is borne out in practice.

4 Trading Redundant Computation for Communication

It is sometimes possible to reduce communication by performing redundant computation. Consider the case where opposing units u and v are on adjacent hexes which happen to be assigned to different processors. If u resides on a hex assigned to processor $P(u)$, then $P(u)$ is responsible for computing the losses that u inflicts on v. But processor $P(v)$ also holds a copy of u, and could do the computation itself, relieving $P(u)$ from the task of communicating the damages u inflicts on v. $P(u)$ will still simulate u attacking v, in order to keep

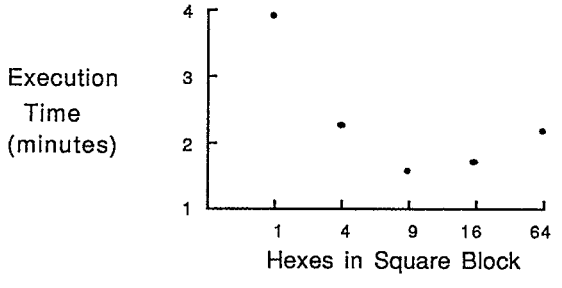

Figure 3: Performance as Function of Granularity

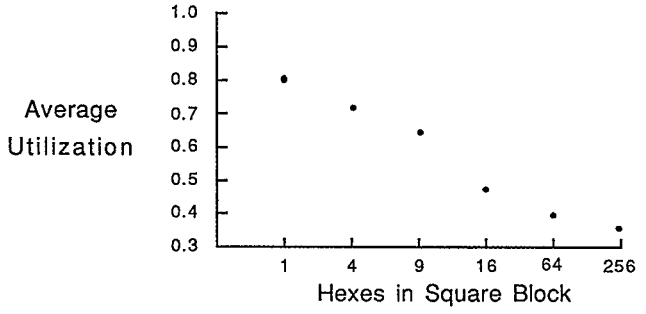

Figure 4: Effects of Granularity on Load Balance

v's state up to date. Thus redundant computation (simulating u's attack on v) can avoid some communication. This tactic may prove to be especially important if communication is very expensive relative to computation. In our experiments we found that performance *suffered* using this technique, largely because a combat's computation is more expensive than the communication of its results. However, we also allowed redundant computation of a unit's new position during the movement activity—the processor owning the unit computes the new position, as does any processor holding a hex adjacent to u's. In this case, the computational cost of movement was dominated by its communication cost, so that redundant computation improves performance.

5 Deadlock Avoidance

Much of the early work in distributed simulations was devoted to the development of synchronization protocols which avoided deadlock [1,6,14]. This work considered the possibility of deadlock occurring in *discrete-event* simulations, due to the inter-process synchronization necessary to insure the simulation's correctness. This type of synchronization problem does not exist in time-driven simulations. Nevertheless, deadlock can occur when there is distributed contention for message buffers.

Consider the following scenario. A message to processor P_i appears in P_i's incoming message queue; the message reports that damages have been inflicted on a unit u in one of P_i's subregions by units in P_j's subregions. As illustrated in Figure 5, when P_i consumes this message it can trigger messages to processors P_k and P_l advising them of u's new status. For P_i to due so, P_k and P_l must both have available space to store incoming messages—if not, the messages from P_i cannot be sent until space is free, and so P_i must block itself. However, either processor P_k or P_l may be blocked for similar reasons, permitting the insidious deadlock cycle to form.

One "solution" to this problem is to simply have an overabundance of message buffer space available. This will not guarantee that deadlock cannot not occur, but can make the probability of deadlock low. On the other hand, it is not difficult to insure that deadlock does not occur in our implementation of Zipscreen. Every message in Zipscreen is either an *original* or a *propagated* message. In the example above, the message from P_j to P_i was original, while the messages to P_k and P_l were propagations of that original message. The problem arose because consumption of the original damages message required the use of buffer space elsewhere (even freeing the buffer space occupied by the original message does not solve the problem since the freed space can be filled while the processor is blocked). Every original message has the potential to do this. However, propagated messages do not further propagate; the consumption of a propagated message will never cause a processor to block.

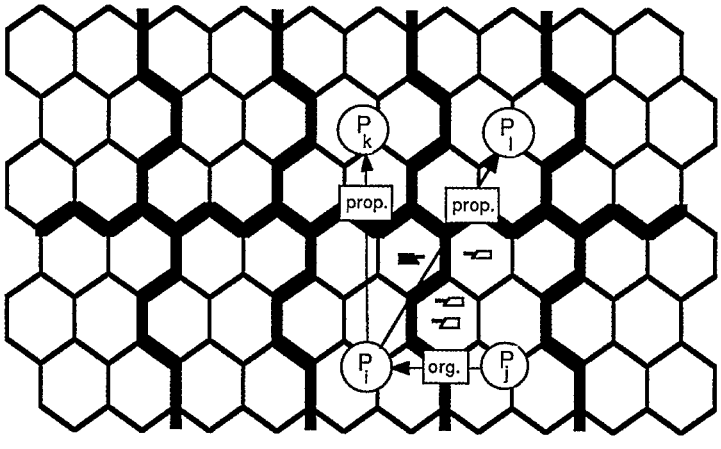

Figure 5: Original Message Causes Propagating Messages

Because of the regularity of our mapping, we know that one processor never communicates with more than six others. [1] It is therefore practical for a processor to reserve message buffer space for every communicating neighbor in order to store incoming original messages, and to reserve space for every processor who might cause a propagation message to be sent (this processor need not be a neighbor, e.g. P_j and P_k in Figure 5). In the unlikely event that the hex blocks are single hexes, space for at most eighteen processors must be reserved. The reservation requirements decrease rapidly as the size of the blocks increase—space for only eight processors is needed using 2×2 hex blocks. Whenever processor P_i sends a message to P_j it includes the status of P_j's original message buffer space, and it includes the status of all propagation buffers reserved for neighbors of P_j. P_j may pick off the buffer status information without actually consuming the message which carried it. The status of P_i's reserved buffers may also be queried (through a reserved query buffer) at any time. Note that the "status" of a buffer may be the number of free bytes, allowing several unconsumed messages to concurrently exist there.

A number of actions may cause a processor to generate an original message to a neighbor. For example, the damage message in the example above is caused by P_j's action of reporting the accumulated damages it has inflicted on unit u. Freedom from deadlock is insured if (i) a processor never takes an action until there is space available to receive all original and propagated messages the action may cause, and (ii) an original message is never consumed until there is space available to receive all propagated messages that the consumption may cause. A processor may consume a propagated message at any time. To show that deadlock cannot occur, note first that if a deadlock cycle forms, then eventually all propagation messages will be consumed. This allows every processor in the deadlock cycle to consume an original message (possibly generating more propagation messages, but these can always be consumed); the consumption of original messages frees the processors to take actions, because the necessary original and propagated message buffer space becomes available. Consequently deadlock never occurs.

6 Summary

An effective parallel execution of domain-oriented time-driven simulations requires the solution to a number of performance problems. First, the simulation workload must be well mapped to keep the load balanced and the communication needs low. We illustrate an effective solution to this problem using a battlefield simulation as a model problem. In the face of significant communication costs it may be advantageous to perform redundant computation to forestall communicating the results of that computation. This point was also illustrated in the model problem. Finally, even though a time-driven simulation does not suffer from the synchronization problems that plague distributed discrete-event simulations, deadlock can still occur. We showed how deadlock can occur in the model problem, and outlined an efficient method of deadlock avoidance.

Acknowledgements: Thanks are due to John Gilmer who provided us with his Zipscreen source code, and to Frank Willard who did most of the early coding. This project has benefited greatly from discussions with Joel Saltz and Paul Reynolds.

[1] This observation assumes that a unit cannot "skip" over a hex from one time-step to the next.

References

[1] K. M. Chandy and J. Misra. Distributed simulation: a case study in design and verification of distributed programs. *IEEE Trans. on Software Engineering*, 5(5):440–452, September 1979.

[2] J.B. Gilmer. *Documentation, State-Space Reconciliation Version of the Zipscreen Prototype Simulation.* Technical Report, BDM Corporation, 1986.

[3] J.B. Gilmer. *Statistical Measurements of the CORBAN Simulation to Support Parallel Processing.* Technical Report BDM/ROS-86-0326, BDM Corporation, 1986.

[4] J.B. Gilmer and J.P. Hong. Replicated state-space approach for parallel simulation. In *Proceedings of the 1986 Winter Simulation Conference*, Washington, D.C., 1986.

[5] D. R. Jefferson. Virtual time. *ACM Trans. on Programming Languages and Systems*, 7(3):404–425, 1985.

[6] P. F. Reynolds Jr. A shared resource algorithm for distributed simulation. In *Proceedings of the Ninth Annual International Computer Computer Architecture Conference*, pages 259–266, Austin, Texas, April 1982.

[7] D. M. Nicol. *The Performance of Synchronizing Networks.* Master's thesis, Department of Computer Science, University of Virginia, January 1984.

[8] D. M. Nicol and F. H. Willard. Problem size, parallel architecture, and optimal speedup. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 347–354, St. Charles, Illnois, 1987.

[9] D.M. Nicol. *Mapping Domain-Oriented Time-Driven Simulations onto Message-Passing Parallel Architectures.* Technical Report 87-51, ICASE, September 1987. submitted to the 1988 SCS Conference on Distributed Simulation, San Diego.

[10] D.M. Nicol and P.F. Reynolds Jr. *Optimal Dynamic Remapping of Parallel Computations.* Technical Report 87-49, ICASE, July 1987. Submitted for publication.

[11] D.M. Nicol and J.H. Saltz. *Dynamic Remapping of Parallel Computations with Varying Resource Demands.* Technical Report 86-45, ICASE, July 1986. to appear in *IEEE Transactions on Computers.*

[12] D.M. Nicol and J.H. Saltz. *Optimal Pre-scheduling of Problem Remappings.* Technical Report 87-52, ICASE, September 1987. submitted for publication.

[13] D.M. Nicol and J.H. Saltz. *Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors.* Technical Report 87-39, ICASE, July 1987. submitted for publication.

[14] J. K. Peacock, E. Manning, and J. W. Wong. Synchronization of distributed simulation using broadcast algorithms. *Computer Networks*, 4:3–10, 1980.

[15] D. A. Reed, L. M. Adams, and M. L. Patrick. Stencils and problem partitionings: their influence on the performance of multiple processor systems. *IEEE Trans. on Computers*, C-36(7):845–858, July 1987.

Biography: David Nicol received the B.A. in mathematics from Carleton College in 1979, and the M.S. and Ph.D. degrees in Computer Science from the University of Virginia in 1983 and 1985. He was a programmer/analyst with Control Data from 1979 to 1982, and a staff scientist with ICASE from 1985-1987. Currently he is an assistant professor of Computer Science at the College of William and Mary. His mailing address is

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