ORIGINAL PAGE IS
OF POOR QUALITY

Mapping a Battlefield Simulation onto Message-Passing Parallel Architectures*

David M. Nicol†
The College of William and Mary

Abstract

Perhaps the most critical problem in distributed simulation is that of mapping: without an effective mapping of workload to processors the speedup potential of parallel processing cannot be realized. Mapping a simulation onto a message-passing architecture is especially difficult when the computational workload dynamically changes as a function of time and space; this is exactly the situation faced by battlefield simulations. This paper studies an approach where the simulated battlefield domain is first partitioned into many regions of equal size, typically there are more regions than processors. The regions are then assigned to processors; a processor is responsible for performing all simulation activity associated with its regions. The assignment algorithm is quite simple, and attempts to balance load by exploiting locality of workload intensity. The performance of this technique is studied on a simple battlefield simulation implemented on the Flex/32 multiprocessor. Our measurements show that the proposed method achieves reasonable processor efficiencies. Furthermore, the method shows promise for use in dynamic remapping of the simulation.

*To appear in the Proceedings of the 1988 SCS Conference on Distributed Simulation, San Diego
†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.

ORIGINAL PAGE IS
OF POOR QUALITY

1 Introduction

Perhaps the most critical performance issue in distributed simulation is that of mapping: without effective mapping policies the speedup potential of parallel processing cannot be realized. Mapping a parallel simulation onto a message-passing architecture is especially difficult when the computational workload dynamically changes as a function of time and space; this is exactly the situation faced by battlefield simulations such as CORBAN [3]. CORBAN focuses on *units* that model military units such as companies, regiments, and battalions. The units move through a two dimensional domain tessellated by hexagons, and engage in combat with "close" units from the opposing side. Figure 1 illustrates a

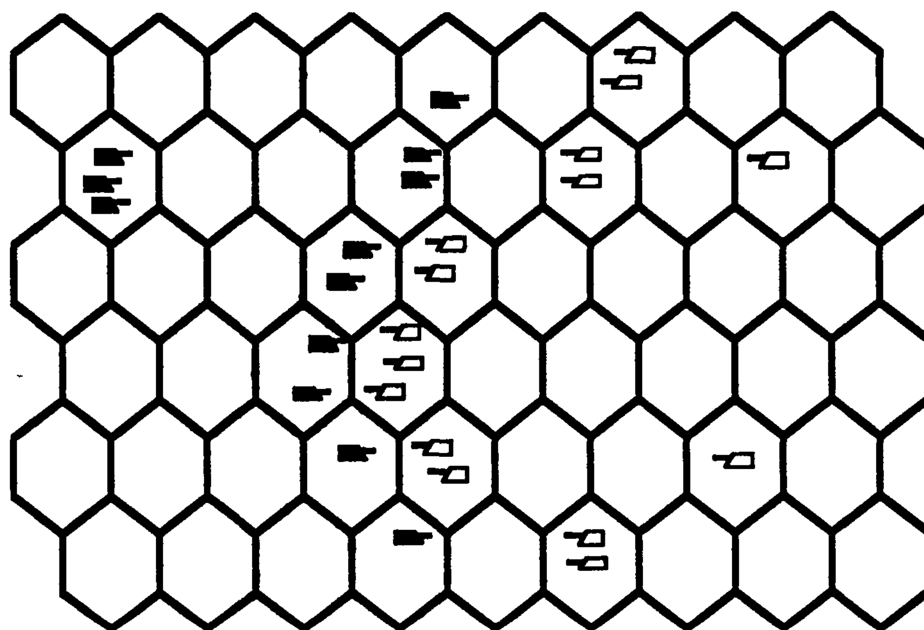

Figure 1: Battlefield Simulation Board

small region of such a domain. Studies have shown that the *perception* of nearby units, and *combat* with opposing units are the most computationally intensive aspect of the simulation[3] .

Both Zipscreen and CORBAN are time-driven, rather than discrete-event simulations. There are strong reasons to suspect that using a discrete-event paradigm for distributed 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 [1,15,13]. A formal treatment in [12] 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 due to context sensitive decision making; this limited ability implies 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". 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.

Because of the relatively high cost of message passing on many 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 mathematical model illustrates that simply distributing the units evenly among processors can lead to serious load imbalance. Suppose that each of N processors is given m units, and each unit independently has a probability p of fighting. Call the number of fighting units in the processor its *load*. Every processor must complete time-step n before any processor can begin time step $n+1$; consequently the time to execute a time-step is the time required by the most heavily loaded processor to execute that step. The processing time of any processor is a random variable; these random variables are independent and identically distributed (with the binomial distribution) with mean $\mu = p \cdot m$ and standard deviation $\sigma = \sqrt{m \cdot p \cdot (1-p)}$. We are interested in $E[M]$, the expected value of the largest of these variables. An approximation in [6] for this quantity is

$$E[M] = \mu + \sigma\sqrt{2\log N}.$$

$E[M]$ can be used to compute *efficiency*: (expected) speedup divided by the number of processors: $e = (Nmp/E[M])/N$. Table 1 tabulates e as a function of p and N when there are 1000 units total, evenly distributed among processors.

We see that performance is poor if the probability of a unit fighting is very low, particularly so if the number of processors is high. Table 1 estimates only the inefficiencies due to load imbalance: the additional overhead due to communication and synchronization will

N	p	e	N	p	e	N	p	e
8	0.5	0.84	16	0.5	0.77	256	0.5	0.37
8	0.25	0.76	16	0.25	0.66	256	0.25	0.25
8	0.1	0.64	16	0.1	0.53	256	0.1	0.16
8	0.05	0.55	16	0.05	0.43	256	0.05	0.12
8	0.01	0.35	16	0.01	0.25	256	0.01	0.05

Table 1: Expected Efficiency

further degrade performance. It is clear from this model that load imbalance is potentially a very serious problem for distributed battlefield simulations.

This paper studies an approach where one assigns pieces of the *domain* to processors, rather than directly assigning units. A processor is responsible for simulating units on its assigned subset of the domain. The simulated battlefield domain is first partitioned into many regions of equal size, typically there are more regions than processors. The regions are then assigned to processors so that an (approximately) equal number of regions are assigned to each processor. The communication requirements of this approach are very local—the vast bulk of communication is between processors holding adjacent regions. At first glance, this technique appears to have virtually the same load balancing philosophy as the previous mapping scheme. However, there is an important difference. The partitioning and assignment of the domain exploits the fact that if there is battle activity in a hex, then there is likely to be battle activity in an adjacent hex. Some degree of load balancing can be achieved simply by assigning adjacent hexes to different processors. This balance comes at the price of increased overhead; some control over the trade-off between load imbalance and overhead is given by parameterizing the partitioning to allow small contiguous groups of hexes to remain on a processor, and assign adjacent groups to different processors. This paper studies the performance of our proposed technique on the Zipscreen simulation, which was modeled after the CORBAN simulation [4].

2 Zipscreen Simulation

Zipscreen [2,4] (developed by the BDM Corporation) is a much simplified version of the CORBAN [3] simulation, developed for the purposes of studying performance issues in mapping battlefield simulations to parallel architectures. Zipscreen focuses on the *perception*, *combat*, and *movement* activities found in CORBAN. During a time-step every Zipscreen unit creates a list of all enemy units on its own hex, and on adjacent hexes. A unit enters combat with units found on this list, calculates losses that it inflicts on enemy units, and reports those losses to the processors owning the afflicted units. At the end of

a time-step, every unit moves, possibly changing hex locations.

The BDM Corporation under contract with DARPA is actively studying issues in the parallel execution of battlefield simulations. In its study of message-passing machines (the Intel iPSC in particular) BDM focused on the perception, combat, and movement activities because of the challenges they present to parallel computing. The efficiency of perception is very much affected by the simulation's distribution across multiple processors, and the cost of inter-processor communication. Combat is computationally intensive, and so has an important influence on load balancing. Inter-unit (and hence potential inter-processor) communication of losses is affected by communication costs as well. The efficiency of movement depends on the cost of communication; the fact that units dynamically move has a profound impact on load balancing and load balancing strategies. Because of these factors, Zipscreen appears to be a good testbed for studying battlefield simulation load balancing issues.

3 Domain Partitioning and Assignment

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

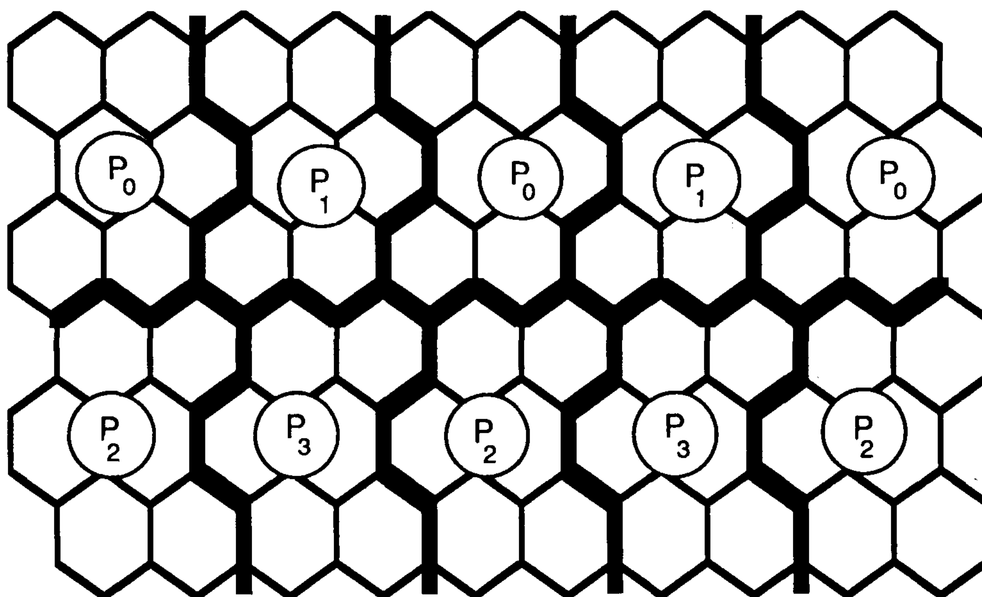

Figure 2: Rectangular Partitioning of Domain

ORIGINAL PAGE IS
OF POOR QUALITY

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 [11,18,7]. 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. Wrapping exploits the observation that for many problems arising from physical domains, workload tends to be positively correlated in space. 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 engagement 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. Simulated combat activity is thus seen to be positively correlated, at least very locally. 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.

The values w and h allow a parameterized partitioning of the domain. Smaller values create a finer granularity of workload, and tend to yield better a balance of load. But because of the communication requirements of maintaining up-to-date unit state information, finer workload granularity requires more communication overhead. 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 [11].

4 Implementation

The organization of the simulation has an important effect on the simulation's performance. This section briefly describes the data structures we used in our implementation of Zipscreen.

BDM made a version of their iPSC source code, written in C, available for use in our load balancing studies. This code is documented in [2]. In adapting their code, we retained the *unit_scoreboard* structure, which describes a combat unit, its identity number, location, direction of movement, speed, side, and number of combat assets. We also retained the *asset_item* structure, which describes the type and strength of a unit's combat asset. We deviate from the original Zipscreen organization in our treatment of hexes. We define a *hex_struct* structure to represent a hex, including such information

as the identities and processor assignments of adjacent hexes. A hex_struct also contains a pointer to a linked list of unit_scoreboard structures representing units resident on the hex. A hex_struct is maintained only for *active hexes*, those with units occupying them; consequently hex_structs are created and deleted throughout a simulation run as units move through the domain. A processor's active hex_structs are organized in a hashtable, with a hex's (one-dimensional) identity number as the hashing key. Naturally, a processor maintains hex_structs for hexes within the processor's assigned regions. In addition, it maintains hex_structs for active hexes which are adjacent to one of the processor's regions, but are assigned to another processor. These *boundary* hex_structs aid in the perception and combat phases of a time-step. The owner of a boundary hex is responsible for sending enough information to keep the description of units on it up to date.

A unit's unit_scoreboard structure can be found in two ways. If the unit's hex location is known, then the hex_struct for that location is quickly found, and the list of units on the hex is scanned. Alternatively, the unit_scoreboard structures are also linked together, allowing an efficient linear scan of the processor's units. The unit_structures list is useful for loops that perform some action (e.g. perception, combat, movement) for every one of a processor's units. The hashtable method is convenient during the perception phase— for a given unit at a known location, we desire a list of all enemy units on the same or adjacent hexes. Units on adjacent hexes are quickly identified since the hex_structs for adjacent hexes are quickly identified.

A time-step in our Zipscreen implementation can be viewed as having a computation phase, followed by a communication phase. Because a processor maintains up-to-date information about active hexes bordering its own regions, it is able to conduct the perception and combat phases without communicating with any other processor. Combat losses which are inflicted on a non-resident unit are simply accumulated by that unit during the computation phase. Once a processor has finished combat it enters the communication phase. First, it sends damage reports to each of its neighboring processors, describing the damage it inflicted on units owned by that processor. Upon receiving a damage report, a processor updates the affected units, and *propagates* the damage report to all of its (other) neighbors who maintain a version of the afflicted unit. A processor receiving a propagated damage report simply updates its copy of the afflicted unit. Once all damage reports have been exchanged, each processor computes the new position of *all* units for which it has a copy. This duplication of computational effort is less expensive than is communicating new positions; this is largely due to the computational simplicity of the Zipscreen movement mechanism. Combat simulations with more realistic (and computationally expensive) movement mechanisms may not find this duplication advantageous. If a unit leaves a hex, the owning processor notifies the new owner (possibly itself); a processor just discards a non-resident unit when that unit changes hex positions. A processor receiving a moved unit notifies all processors adjacent to that unit's hex. A unit leaving the board simply disappears. The time-step is finished when all processors have completed the unit movement activity.

The organization of our version of Zipscreen is similar in philosophy to the *state space replication* method studied by Gilmer and Hong [4]. However, there are important differences between the two methods. The state space replication method requires that every processor maintain in memory an up-to-date version of the entire simulation state space. Clearly, it is quite expensive to maintain the entire state space in every processor, as this requires global dissemination of new data at the end of each time-step. Our method requires that a processor duplicate some of the domain (units on hexes adjacent to its own), but only that portion of the domain which is likely to affect the units implicitly assigned to the processor; the technique of sharing only boundary information is standard practice in the parallel solution of partial differential equations using relaxation methods. Sharing only boundary information limits the amount of communication necessary: using our scheme a processor need communicate with no more than six other processors (although this figure could rise if more complicated movement and rules of engagement were allowed). Our method achieves substantial savings in communication costs over the state space replication method, and should suffer less from load imbalance.

Our version of Zipscreen currently runs on the Flex/32 Multi-computer 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 NSC32032 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. For the purposes of measuring the additional cost of using a message-passing paradigm, a shared-memory version of Zipscreen has also been written and is being implemented on the Flex. In addition, Zipscreen is currently being ported to the Intel iPSC. We intend to use these additional codes to study the impact that architecture (or architectural paradigm) has on performance.

5 Empirical Studies

We ran our code on a variety of battlefield configurations, using a variety of partitionings. The experiments were not intended to closely model realistic battlefield situations; rather, they were constructed to exercise the simulation in those parameters we anticipate will be most influential on realistic simulations. Every run simulates a battle between two sides, each having five hundred units; every run advances fifty time-steps. The two sides are initially separated by an imaginary line that runs through both the top and bottom hex rows. All of a side's units are placed randomly in a corridor a few hexes wide that runs the length of the line. The units are given random speeds and random directions,

Parameter	*Values*
Board Size	32×32, 64×64
Average Unit Speed	0.06 hexes/step (*slow*) , 0.175 hexes/step (*fast*)
Battleline Orientation	Verticle, Diagonal

Table 2: Problem Parameters

with the provision that the direction chosen eventually intersects the separating line (thus ensuring that two sides do approach each other). Our runs varied the board size, the average unit speed, and the angle of the separating line. Table 2 summarizes the values those parameters assumed.

For every boardsize, we tested all possible square power-of-two square partitions, and all possible horizontal strips having a power-of-two number of rows. We used eight and sixteen processors. For each run we measured the elapsed running time (excluding the time required to download the parallel machine). We also measured the serial running time of each configuration, allowing the calculation of a run's efficiency. Tables 3 and 4 tabulate these measurements. *Size* refers to the size of the aggregated domain block, *Procs* to the number of processors used; the running time is given in minutes.

Several important trends are discernible from this data. First of all, this mapping method clearly works—within the allowed family of mappings there are partition sizes which yield reasonably good efficiencies. However, some care needs to be taken to avoid the bad mappings. A second trend is that strip partitions perform better than square partitions. The primary reason for this is lower communication and synchronization costs. Using square partitions a processor must communicate and synchronize with six others; using strips a processor communicates with only two others. If the startup costs of a message are high, then reducing the total number of messages can lead to better performance, even if the *volume* of message traffic is higher. This phenomenon has also been noted in parallel numerical codes [16]. Note however that the good performance for strips heavily depends on proper strip orientation; it also depends on the battleline extending the full height of the board. If the battleline orientation can not be predicted, or if the battle bunches up in one corner, then square partitions will be more effective than strips. A third important trend is that the 32×32 board running times are significantly larger than the 64×64 board running times, and the 32×32 board efficiencies are significantly higher than the 64×64 board efficiencies. The higher running times are due to the fact that there are more interactions between units on the smaller board; the higher efficiencies follow because the average processor communication costs varies very little between the two board sizes (a fact given by additional measurements not shown here). It has been noted time and time again in parallel processing research that the ratio of communication to computation is a strong factor in determining performance. All of the variations in performance shown

32 × 32 Board, Fast, Vertical			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	4.12	0.15
2 × 2	"	2.63	0.24
4 × 4	"	2.43	0.26
8 × 8	"	2.76	0.23
1 × 32	"	1.46	0.44
2 × 32	"	1.42	0.45
1 × 32	8	2.69	0.48
2 × 32	"	2.12	0.61
4 × 32	"	2.30	0.56

32 × 32 Board, Fast, Diagonal			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	3.79	0.15
2 × 2	"	1.79	0.31
4 × 4	"	1.71	0.32
8 × 8	"	2.06	0.27
1 × 32	"	1.31	0.42
2 × 32	"	1.27	0.44
1 × 32	8	2.44	0.45
2 × 32	"	1.82	0.61
4 × 32	"	1.99	0.56

32 × 32 Board, Slow, Vertical			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	4.85	0.20
2 × 2	"	3.59	0.28
4 × 4	"	3.66	0.27
8 × 8	"	3.92	0.25
1 × 32	"	1.87	0.52
2 × 32	"	1.87	0.52
1 × 32	8	3.40	0.58
2 × 32	"	2.88	0.68
4 × 32	"	3.07	0.64

32 × 32 Board, Slow, Diagonal			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	4.09	0.18
2 × 2	"	1.95	0.39
4 × 4	"	2.22	0.34
8 × 8	"	2.44	0.31
1 × 32	"	1.51	0.50
2 × 32	"	1.47	0.51
1 × 32	8	2.76	0.55
2 × 32	"	2.22	0.68
4 × 32	"	2.36	0.64

Table 3: Measurements from 32 × 32 Board

64 × 64 Board, Fast, Vertical			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	3.34	0.12
2 × 2	"	1.45	0.28
4 × 4	"	1.28	0.31
8 × 8	"	1.27	0.32
16 × 16	"	1.40	0.29
1 × 64	"	1.10	0.36
2 × 64	"	0.77	0.52
4 × 64	"	0.79	0.51
1 × 64	8	1.90	0.42
2 × 64	"	1.43	0.56
4 × 64	"	1.18	0.67
8 × 64	"	1.30	0.62

64 × 64 Board, Fast, Diagonal			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	2.63	0.12
2 × 2	"	1.17	0.27
4 × 4	"	0.80	0.39
8 × 8	"	0.87	0.36
16 × 16	"	1.05	0.30
1 × 64	"	0.82	0.38
2 × 64	"	0.60	0.52
4 × 64	"	0.57	0.53
1 × 64	8	1.58	0.39
2 × 64	"	1.08	0.58
4 × 64	"	0.93	0.67
8 × 64	"	0.95	0.66

64 × 64 Board, Slow, Vertical			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	3.44	0.12
2 × 2	"	1.43	0.28
4 × 4	"	1.41	0.28
8 × 8	"	1.40	0.29
16 × 16	"	1.45	0.28
1 × 64	"	1.06	0.37
2 × 64	"	1.43	0.28
4 × 64	"	1.41	0.28
1 × 64	8	1.89	0.42
2 × 64	"	1.47	0.54
4 × 64	"	1.29	0.62
8 × 64	"	1.35	0.59

64 × 64 Board, Slow, Diagonal			
Size	*Procs*	*Minutes*	*Efficiency*
1 × 1	16	2.51	0.07
2 × 2	"	1.08	0.17
4 × 4	"	0.81	0.23
8 × 4	"	0.87	0.21
16 × 16	"	0.97	0.19
1 × 64	"	0.75	0.25
2 × 64	"	0.65	0.29
4 × 64	"	0.59	0.32
1 × 64	8	1.44	0.26
2 × 64	"	0.99	0.38
4 × 64	"	1.05	0.36
8 × 64	"	0.97	0.39

Table 4: Measurements from 64 × 64 Board

ORIGINAL PAGE IS
OF POOR QUALITY

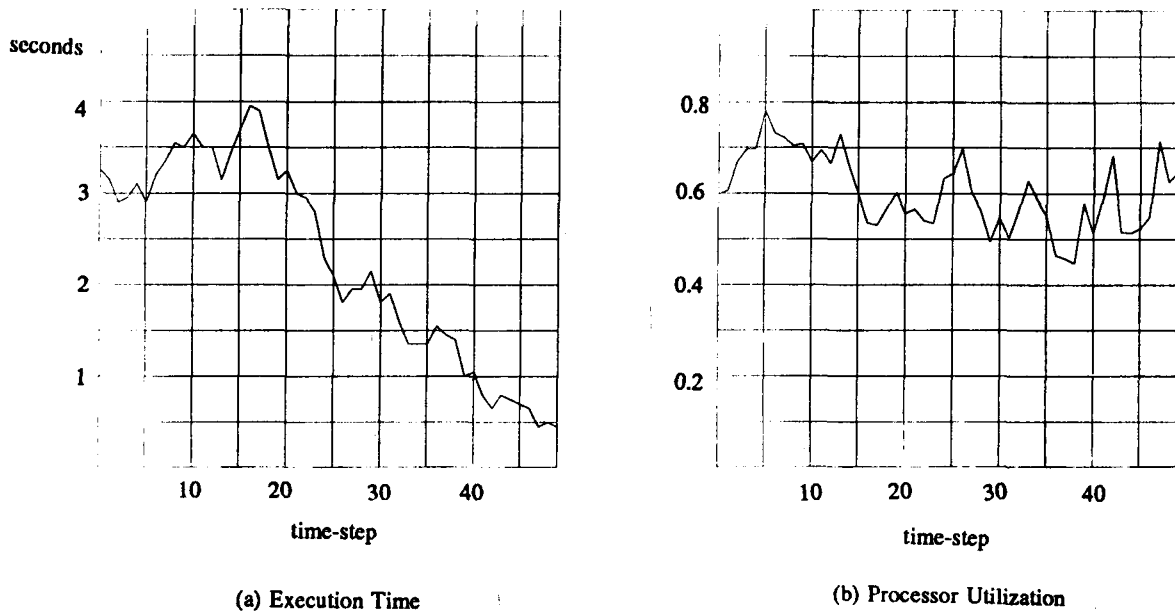

Figure 3: Execution Time and Processor Efficiencies During Perception and Combat

in our data can be explained in those terms. Finding the best mapping for the problem amounts to finding the largest communication to computation ratio.

The organization of our Zipscreen code allows us to measure the time spent in perception and combat, during which time no communication occurs. Consequently, we are able to measure the processor efficiency due solely to load balancing, and compare it in a loose way to the efficiencies given in table 1. For one example run (using a 32×32 board, a 2×2 partition, a diagonal separating line, sixteen processors, and fast unit speeds), figure 3(a) plots the longest time any processor spends in perception and combat as a function of time-step; 3(b) plots processor efficiency as a function of time-step for that same run. Despite the fact that the fraction of combating units falls off dramatically to less than 5% by the end of the run, the processor efficiencies are almost always better than 50%, they are often better than 60%. This compares favorably with the "random scatter" efficiencies predicted in table 1. This is very clear evidence that the load balancing method is working. As noted before, our method has the additional advantage of limiting a processor's communication to a few logical neighbors. On hypercube architectures such as the Intel iPSC [14], it is possible to assign logical processors to physical processors so that any message between two logical neighbors crosses no more than two communication links.

Large blocks reduce the the amount of communication at the risk of load imbalance, and small blocks balance load at the price of increased communication volume. The very

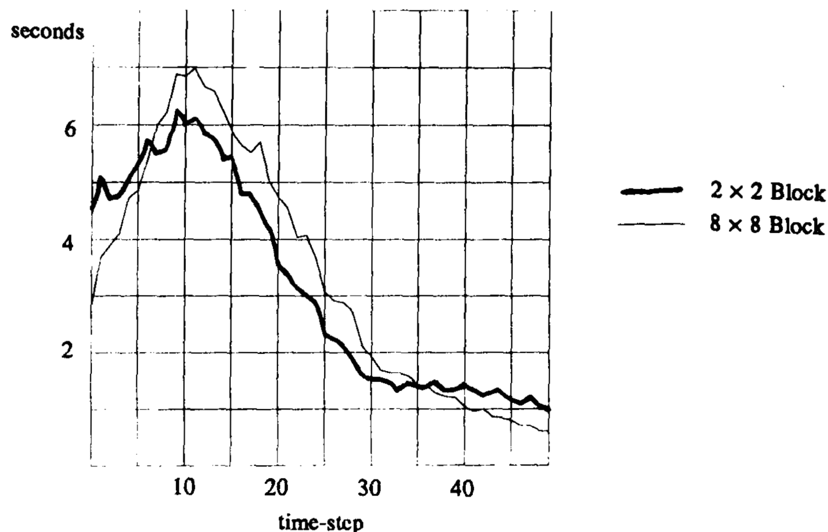

Figure 4: Running Times for Two Different Partitions

dynamic nature of simulations can create the situation where in some periods there is intense computational activity, while in others there is relatively little. It can be advantageous to use large blocks during low activity periods to minimize communication, and to use small blocks during high activity periods to better balance the load. This is the case using "slow" units—there is an initial period of low computational activity as the two sides approach. The computational workload increases dramatically when the two sides are throughly mixed, and tails off as they pass through each other. Figure 4 illustrates two superimposed plots of the execution time versus time-step. One line plots the performance of a 2×2 partition on a 32×32 board, the other plots the performance of a 8×8 partition. We see an obvious potential for dynamically remapping the simulation in order to find the best partition-to-problem fit at every time-step. Dynamic remapping is essential if we use strip partitions, and then the battleline moves to a configuration which defeats the strip orientation. Of course, dynamically remapping the simulation exacts a delay which must be considered. Some solutions to the dynamic remapping problem are given in [9,17,8,10]; we intend to study these techniques on the battlefield simulation problem.

ORIGINAL PAGE IS
OF POOR QUALITY

6 Summary

Early inquiries into the suitability of message-passing architectures for distributed battlefield simulation have shown that load-balancing is a serious problem. Load imbalance arises from the fact that the behavior of the simulation workload is unpredictable, and message-passing architectures seem to require semi-static mappings of workload to processors. As the workload changes, any initial balance deteriorates, and performance is dominated by the most heavily loaded processor. We are studying an approach which attempts statistically to balance load. Under our approach we can chose an appropriate degree of granularity, which in turn controls trade-off between good (probable) load balance, and communication overhead. We have implemented the Zipscreen simulation, have shown empirically that our approach is viable, and have demonstrated the trade-off between load balance and communication inherent in our approach. Future work will investigate related mapping strategies, including dynamic remapping, and will consider the implications of using our approach on more realistic simulations such as CORBAN.

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.

References

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

[2] GILMER, J. *Documentation, State-Space Reconciliation Version of the Zipscreen Prototype Simulation.* Tech. Rep., BDM Corporation, 1986.

[3] GILMER, J. *Statistical Measurements of the CORBAN Simulation to Support Parallel Processing.* Tech. Rep. BDM/ROS-86-0326, BDM Corporation, 1986.

[4] GILMER, J., AND HONG, J. Replicated state-space approach for parallel simulation. In *Proceedings of the 1986 Winter Simulation Conference* (Washington, D.C., 1986).

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

[6] KRUSCAL, C. P., AND WEISS, A. Allocating independent subtasks on parallel processors. *IEEE Trans. on Soft. Eng. SE-11*, 10 (October 1985), 1001–1015.

[7] MORRISON, R., AND OTTO, S. *The Scattered Decomposition for Finite Elements.* Tech. Rep. 286, Caltech Concurrent Computing Project, May 1985.

[8] NICOL, D., AND REYNOLDS, JR., P. *Optimal Dynamic Remapping of Parallel Computations.* Tech. Rep. 87-49, ICASE, July 1987.

[9] NICOL, D., AND SALTZ, J. *Dynamic Remapping of Parallel Computations with Varying Resource Demands.* Tech. Rep. 86-45, ICASE, July 1986. To appear in *IEEE Transactions on Computers.*

[10] NICOL, D., AND SALTZ, J. *Schedules for Mapping Irregular Parallel Computations* . Tech. Rep. 87-52, ICASE, September 1987.

[11] NICOL, D., AND SALTZ, J. *Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors.* Tech. Rep. 87-39, ICASE, July 1987.

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

[13] PEACOCK, J. K., MANNING, E., AND WONG, J. W. Synchronization of distributed simulation using broadcast algorithms. *Computer Networks 4* (1980), 3–10.

[14] RATTNER, J. Concurrent processing: a new direction in scientific computing. In *AFIPS Conference Proceedings, National Computer Conference* (1985), pp. 157–166.

[15] REYNOLDS, JR., P. A shared resource algorithm for distributed simulation. In *Proceedings of the Ninth Annual International Computer Computer Architecture Conference* (Austin, Texas, April 1982), pp. 259–266.

[16] SALTZ, J., NAIK, V. K., AND NICOL, D. Reduction of the effects of the communication delays in scientific algorithms on message passing mimd architectures. *SIAM J. Sci. Stat. Comput 8*, 1 (1987), s118–s134.

[17] SALTZ, J., AND NICOL, D. Statistical methodologies for the control of dynamic remapping. In *Proceedings of the ARO Conference on Medium Scale Multiprocessing* (Stanford University, 1986).

[18] WON, Y., AND SAHNI, S. Maze routing on a hypercube multiprocessor computer. In *Proceedings of the 1987 International Conference on Parallel Processing* (St. Charles, Illinois, August 1987), pp. 630–637.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA CR-178396 ICASE Report No. 87-51		

4. Title and Subtitle	5. Report Date
MAPPING A BATTLEFIELD SIMULATION ONTO MESSAGE-PASSING PARALLEL ARCHITECTURES	October 1987
	6. Performing Organization Code
7. Author(s)	8. Performing Organization Report No.
David M. Nicol	87-51
	10. Work Unit No.
	505-90-21-01
9. Performing Organization Name and Address	11. Contract or Grant No.
Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665-5225	NAS1-18107
	13. Type of Report and Period Covered
12. Sponsoring Agency Name and Address	Contractor Report
National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225	14. Sponsoring Agency Code

15. Supplementary Notes

Langley Technical Monitor:
Richard W. Barnwell

Submitted to Proc. of SCS Conference on Distributed Simulation

Final Report

16. Abstract

Perhaps the most critical problem in distributed simulation is that of mapping: without an effective mapping of workload to processors the speedup potential of parallel processing cannot be realized. Mapping a simulation onto a message-passing architecture is especially difficult when the computational workload dynamically changes as a function of time and space; this is exactly the situation faced by battlefield simulations. This paper studies an approach where the simulated battlefield domain is first partitioned into many regions of equal size, typically there are more regions than processors. The regions are then assigned to processors; a processor is responsible for performing all simulation activity associated with its regions. The assignment algorithm is quite simple and attempts to balance load by exploiting locality of workload intensity. The performance of this technique is studied on a simple battlefield simulation implemented on the Flex/32 multiprocessor. Our measurements show that the proposed method achieves reasonable processor efficiencies. Furthermore, the method shows promise for use in dynamic remapping of the simulation.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
simulation, parallel processing, multiprocessors, message-passing	61 - Computer Programming and Software Unclassified - unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	17	A02

NASA FORM 1626 OCT 86