*Proceedings of the 1994 Winter Simulation Conference*
*ed. J. D. Tew, S. Manivannan, D. A. Sadowski, and A. F. Seila*

# MASPAWS - A MASSIVELY PARALLEL WAR SIMULATOR

N. Chandrasekharan
and
Zhi-Hai Ma
Dept. of Mathematical Sciences
Loyola University of Chicago
Lake Shore Campus
Chicago, IL 60626, U.S.A.

Udaya B. Vemulapati
and
William J. Porthouse, Jr.
Dept. of Computer Science
University of Central Florida
Orlando, FL 32816, U.S.A.

Allen T. Irwin
SAIC
Technology Parkway
Orlando, FL 32826, U.S.A.

## ABSTRACT

We describe the design, implementation and performance of a battlefield simulator for massively parallel SIMD machines. MasPaWS (Massively Parallel War Simulator) is a first generation prototype that simulates the battle dynamics of a two-dimensional theater to the level of individual tanks in a battlefield having certain terrain features. Objects in the battlefield are the terrain features and tanks. The battle cycles through a set of protocols: *perception* and *combat*, *update combat*, *migration* and *update migration*. MasPaWS, written in Maspar Programming Language (MPL), has been implemented on MasPar machine. The salient features of MasPaWS include: (1) use of simple protocols to match machine architecture, (2) parametrization of several quantities to control the computation of and communication between MasPar processors and to estimate performance, and (3) interactive user input for terrain specification and generation and other key parameters. Our work appears to be the first of its kind in war simulation on a massively parallel SIMD machine, using thousands of processors. Our results indicate that MasPaWS is both data and architecture scalable for battlefield simulation, while achieving high efficiency at the same time.

## 1 INTRODUCTION

Computer simulation has been a major tool for analysis, prediction and training in the last several years. The increasing computational complexity of simulating physical systems has led researchers to recognize the use of parallel computers. Computer simulations which use many processors have been variously called as *distributed*, *parallel*, *concurrent*, etc. We will use, for the purposes of this work, *parallel simulation* to represent discrete event time-driven simulation carried out on tightly-coupled multiprocessors. For a general discussion on parallel simulation, the reader may consult Fujimoto (1990) and Prasad (1990). For a more recent survey on parallel simulation, including battlefield simulation, we refer to Nicole (1994).

### 1.1 Battlefield Simulation

Battlefield simulations are amongst the most irregular, computationally intensive, and complex simulations in existence (Weiland *et al.* 1989). A central issue in parallel simulation is *work-to-processor mapping*, especially where the computational load dynamically varies with both time and space. Parallel battlefield simulators use either a *functional* or *spatial* mapping of work to processors. Functional mapping is achieved by distributing battle units to processors where each processor is responsible for a certain number of operational combat units (Gilmer and Hong 1986). Spatial mapping is carried out by tessellating the battle theater into regular geometric objects (rectangles, hexagons, etc.) and assigning processors to fixed regions of the battlefield (Nicole 1987, Nicole 1988, Prasad 1990, Gilmer 1988). In both scenarios, as battle progresses, *load balancing* becomes a serious issue. One way of coping with load balancing is to introduce dynamic assignment of work. But this has to be weighed against the cost of determining and executing a new assignment at various points in simulation. Another key issue in parallel simulation is *communication overhead*. In functional mapping, any two arbitrary processors may be required to communicate battle data whereas in spatial mapping, communication is usually confined within a certain neighborhood of processors. Where communication overhead between processors increases with distance between them, there is a good reason to perform spatial mapping. Hence existing parallel battlefield simulators on distributed-memory, parallel machines (e.g., hypercubes) use spatial mapping (Deo, Medidi and Prasad 1992, Nicole 1987). We now discuss some of the previous works on battlefield simulation.

### 1.2 Previous Work On War Simulation

One of the influential works in battlefield simulation is Quickscreen (Hawkins and Thompson 1985) developed by the BDM corporation for the US Army, which was later renamed as CORBAN (Corps Battle Analysis). Quickscreen was designed to run on a uniprocessor machine using FORTRAN. The simulation is time-stepped, of Corps-scope, of battalion resolution and could simulate up to 700 battle units with terrain effects. Zipscreen, a time-stepped simulator representative of the structure of CORBAN, was implemented on BBN Butterfly, a shared memory parallel machine (Gilmer 1988). The simulator worked on a battlefield decomposed into hexagons and attained efficiencies of 40%–75% on 124 processors with the number of combat units ranging from 480 to 800. A version of Zipscreen was implemented on Flex/32 multicomputer with 20 processors (Nicole 1987) assuming battalion sized units on a hexgonally partitioned battle terrain. The typical speedup attained was 8.5 on 16 processors, resulting in 53% efficiency. Weiland *et al.* (1989) describe a combat simulation called the Concurrent Theater Level Simulation designed at the Jet Propulsion Laboratory. This was implemented on the Caltech/JPL Mark III Hypercube and the BBN Butterfly machine. In this event-stepped simulator, a speedup of 28.6 on 60 Mark III processors (48% efficiency) and 36.8 on 100 BBN Butterfly processors (36.8% efficiency) was attained. More recently, Deo *et al.* (Deo, Medidi and Prasad 1992) describe research into processor allocation under three different schemes for a time-driven battlefield simulation. In all of the 3 simulators, the combat units are of battalion resolution. The first scheme uses static allocation of the battlefield domain to processors and with 50 combat units on each side, a speedup of 9.5 was observed on a 16 node Intel hypercube iPSC/1 computer. In the second scheme, the above code ported on a BBN Butterfly GP 1000 machine recorded a speedup of 8 using 16 processors. The third method uses a dynamic processor allocation strategy which resulted in a speedup of 12 on 16 processors and 17 on 27 processors. Another work worthy of mentioning in the sequential setting is GISMO (Game for Intelligent Simulated Military Opponents) (Van Brackle 1992). This is an interactive software testbed in which simulated forces may be compared to one another.

### 1.3 Present Work

We observe that all of the above simulators have been designed at a coarse resolution (battalion level, usually) and implemented on MIMD (Multiple Instruction Multiple Data) machines. The efficiencies of the simulators range from 35% to 75%, on processors numbering from 16 to 124. Here, we describe the design, development and study of MasPaWS (Massively Parallel War Simulator) on a SIMD (Single Instruction Multiple Data) machine (MasPar) using several thousands of fine-grained processors. Our resolution of the battle units is to the level of individual tanks each having a gun with reloadable supply of ammunition. Briefly, the battle terrain is decomposed into squares $m$ rows long and $n$ columns wide, where $m \times n$ is the size of the processor grid of the MasPar parallel machine, resulting in a simple static mapping. The objects in the simulation are *tanks* and *terrain effects* such as forest, mountain, water and plain. Each processor is responsible for the actions happening in its battle square. Actions include perceiving for an enemy tank within a certain *depth* of squares in front of each tank under its control, performing line-of-sight computations taking into account the terrain effects, engage enemy tanks in battle, update casualty information, plan movement of its combat vehicles, and move each of its tank to a possible new position avoiding collisions. The simulation cycles through the above protocols and collects battle statistics.

#### 1.3.1 MasPaWS: Salient Features & Results

- MasPaWS is a first generation prototype of simulating a battlefield on a SIMD machine and appears to be the first of its kind.
- MasPaWS supports automatic generation of terrain data for rectangular terrains. Further, it provides three different runtime codes (1) for timing purposes only, (2) for logging important events in the battle, and (3) for debugging purposes using asserts.
- It provides several parametrized variables which control the simulation to study various effects. Parameters include GunRange, TankSpeed, Assets, ReloadTime, GridSize, etc.
- Experiments suggest the MasPaWS is both *data* and *architecture* scalable, in a sense we describe later.
- Our performance studies of MasPaWS indicates that its efficiency was over 80% consistently.

To facilitate expressing the performance of a SIMD machine we now introduce some relevant terminology. Let $t$ be the parallel run-time of a parallel program on a SIMD machine with $n$ processors for a certain input data. Then we define the total work done, $W$, by the program to be $W = t \times n$ for that input data. Let $t_{min}$ be the time taken for the program on a certain data set for the smallest configuration of processors available, $P_{min}$. ($P_{min} = 1K$ processors in the case of MasPar). Let $t_{current}$ be the time taken for the program on the same data set on the number of processors under consideration $P_{current}$. It may be noted that for a fully configured MasPar, $P_{current}$ can be varied as 1K, 2K, 4K, 8K and 16K processors. Then we define *speedup*

$$S = t_{min}/t_{current}$$

and *efficiency*

$$E = \frac{W_{min}}{W_{current}} = \frac{t_{min} \times P_{min}}{t_{current} \times P_{current}}$$

Note that the above definitions are consistent with the well-known definitions of speedup and efficiency for MIMD systems.

### 1.4 MasPar Overview

The MasPar parallel processing system is a massively parallel SIMD (Single Instruction, Multiple Data) machine. The system is available in 1K, 2K,···, 16K processors with maximum memory of 64K in MP-1 or 256K in MP-2, per processor element (PE). MP-1 is rated at a peak 175 MFlops and MP-2 at 400 MFlops for 1K processors. The machine has a DECstation 5000 as a frontend host. The backend, known as the Data Parallel Unit, has an Array Control Unit (ACU) and the Processor Element (PE) Array. The ACU drives the PE array unit execution by broadcasting instructions. The processors are interconnected to form a grid-like configuration with the boundary processors wrapped around toroidally. Two types of communication between processors are possible: *xNet* and *router*. A processor can communicate with a processor at a distance $i$ away by an xNet[$i$] communication in one of eight directions (North, South, East, West, Northeast, Northwest, Southeast, and Southwest) directly or by means of a special router hardware. Generally speaking, xNet is used for short-distance and router for long-distance communications for efficient communication.

The rest of the paper is organized along the following lines. In the next section, we provide the background needed for understanding the design features of MasPaWS along with relevant data structures. Section 3 deals with the detailed description of MasPaWS protocols. The experimental performance of MasPaWS under varying conditions and parameters is presented in Section 4, followed by conclusions.

## 2 TECHNICAL BACKGROUND OF MASPAWS

### 2.1 Conflict Environment

The battlefield has two types of grids points, viz., *level-0* and *level-1*. A level-1 grid is made up of $N \times N$ level-0 grid points, where $N$ is parametrized as GridSize in the implementation. Hence a level-0 grid point is terrain at its finest resolution and uniquely specified by coordinates $(x, y)$. For the sake of convenience, we will call a level-0 grid point as a "point". Each point is associated with a terrain feature, which is one of *plain*, *forest*, *water* and *mountain*. Terrain features affect line-of-sight and hence perception and movement of vehicles. Each point has only one feature and at most one tank (friendly or enemy).

### 2.2 Combat Vehicle

Tanks are the only fighting vehicles and each tank has a gun. A tank can move in one of eight directions East, West, North, South, NorthEast, NorthWest, SouthEast and SouthWest and rotate on its vertical axis. The tanks have three speeds: Forward, Forward-Half and Halted. A tank can reverse direction by turning 180°. The speed of a tank is affected by the terrain it is currently in and moving into. The tanks turn differently at different speeds. Each tank has a limit of certain rounds of ammunition. After a tank runs out of ammunition, reloading it with ammunition involves a certain time delay. The gun has a certain range, and when it fires at a target the damage done obeys a certain probability function depending on parameters like speeds of the tanks, distance, etc. Tanks operate independently. No command and control or organizational structure is simulated in this initial version. Representation of tactics is restricted to requiring each tank to move toward the greatest perceived threat. The battlefield is rectangular and it is divided into a certain number of level-1 grids depending on the user input. Each processor is responsible for the battle actions of the level-1 grid it is assigned.

### 2.3 Battle Parameters

To facilitate a detailed study of the simulation, we have parametrized as many variables as possible which arise in the conflict environment and vehicular characteristics. These are provided below.

#### 2.3.1 Global Variables

`GridSize` defines the size of the battlefield. GridSize*GridSize equals number of level-0 points making up a level-1 grid. `UnitSize` is the maximum number of (live) tanks (for each color) that can be present in a level-1 grid. `FullTankSpeed` is the number of level-0 points the tank can move at full speed. `GunRange` is the range of the gun in number of level-0 points. `Assets` denotes the initial number of rounds of ammunition in the tank. `TankStrength` is the number of *hits* needed to destroy a tank. `PerceptionDepth` denots the number of level-1 grids the tank can perceive in any direction. `BlueArmyStrength` and `RedArmyStrength` indicate the number of tanks in the blue army and red army respectively. `SimulationCycleLength` is the number of cycles of simulation where each cycle consists of all the protocols namely, perception and combat, combat update, migration and migration update. `CombatCycleLength` is the number of cycles involving only perception and combat actions. `MigrationCycleLength` is the number of cycles involving only migration actions. `ReloadTime` is the time units taken for reloading the ammunition in a tank. `BattlefieldLength` is the length of the battlefield in terms of number of level-1 grids. `BattlefieldWidth` is the width of the battlefield in terms of number of level-1 grids. `NoOfGrids` is the number of level-1 grids that have tanks initially. *NoOfTerrainBlock* is the number of terrain blocks having special terrain features. The terrains are assumed to be rectangular in shape.

### 2.3.2 Data Structures

The following data structures used in MasPaWS will be needed to describe the protocols in Section 3.

Tank Descriptor includes following fields: `Speed`, `AssetsLeft`, `CurrentStrength`, `TankId`, identification number for a specific tank, `TankPosition`, the level-0 point that the tank is on, `Direction`, and `Status`. If `Status` is positive, it denotes the number of time units still needed for a complete reload. Normally it is zero.

The data structure *Hit* is local to every PE and records information about combat and it has the following fields: `GridId`, PE of the level-1 grid that holds a tank, `FoeTankId`, the id of the tank, `hits`, the no. of *hits* suffered by the tank, and `FoeTankColor`, the color of the tank.

In addition, each PE has the following variables. `NoOfBlueTanks` is the number of blue tanks on this level-1 grid. `NoOfRedTanks` is the number of red tanks within this level-1 grid. `TerrainView` is a 2-D array of the struct TerrainInfo of all level-0 points in a level-1 grid. TerrainInfo has a *TerrainValue* field and a *TankPresent* field. Terrain value is one of plain, forest, mountain and water. TankPresent is 1 when a tank is on that point and 0 otherwise. `RedTankInfo` is an array of red tank descriptors. `BlueTankInfo` is an array of blue tank descriptors. `HitListSize` is the number of tanks engaged. `HitList` is an array of data structure tt Hit.

## 3 DESCRIPTION OF MASPAWS PROTOCOLS

### 3.1 Perception and Combat Protocol

For each level-1 grid in parallel and for each of local tank $t_i$ that has ammunition the following steps are carried out:

1. Search from near to far and from left to right, each level-1 grid within the PerceptionDepth for an enemy tank $t_j$, in the facing direction of $t_i$. In each grid, pick an enemy tank in the order of how the tanks are listed. The reader may note that for the purposes of the simulator, an *engagment* of a tank by another is different from a *hit.*

2. If $t_j$ is visible to $t_i$ and $t_i$ is the first tank to engage $t_j$, then $t_i$ will engage $t_j$. Otherwise, check the HitList to see if $t_j$ was hit before. If it was, pick a random number $p$ between 0 and 1. If $p$ is larger than a certain threshold, $t_i$ engages $t_j$ and if it is a hit, update the HitList. If $p$ is smaller than the threshold then $t_i$ looks at the next enemy tank. If $t_j$ is not on the HitList, $t_i$ engages $t_j$ and puts $t_j$ onto Hitlist. If $t_j$ is not visible, then search for the next enemy tank.

3. During engagement, first calculate the euclidean distance $D$ between two tanks. Then find the speed of both the local tank and the remote tank and decrease the AssetsLeft by 1. Use the formula

$$P = Round(100 \times e^{-3\times(GunRange)^{-4}\times D^2} - 5),$$

to find the hit probability $P$ (expressed in percentage). Then modify $P$ using the following rules:

if local tank is at half speed, decrease $P$ by 3; if local tank is at full speed, decrease $P$ by 5; if remote tank's speed is half, decrease $P$ by 5; if remote tank's speed is full, decrease $P$ by 10. Then pick a random number $R$ between 0 and 100. If $R < P$ then it is a hit, else it is not. The above formulas are based on the work of Van Brackle (1992).

4. To check visibility the following **line-of-sight calculation** is used. Let $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ be two points on the battle grid. The line-of-sight between $P_1$ and $P_2$ should involve all of the intermediate points lying on the straight line connecting these two points. The closest approximation to this on a grid is given by the following simple procedure: Let $x = |x_2 - x_1|$ and $y = |y_2 - y_1|$. Let

$g = \gcd(x, y)$. Let $\delta x = x/g$ and $\delta y = y/g$. Then line-of-sight from say, $P_1$, is determined by checking all of the grid points obtained by iteratively incrementing (or decrementing) $x_1$ by $\delta x$ and $y_1$ by $\delta y$. The *gcd* is calculated by a fast Euclid algorithm.

5. The casualty update is performed by each processor running down the HitList of a remote grid and updating the cumulative hits. More details can be found in Chandrasekharan and Vemulapati (1994).

## 3.2 Migration Protocol

Tanks will gather information about the number of friend and foe tanks within their *PerceptionDepth* and set their *EnemyDirection* to where they are needed most. Tanks will then *migrate* in that direction while avoiding other tanks (both alive and dead), and hazardous terrain features (e.g., mountains, water, and battlefield borders). The Migration Protocol can be further broken down into two smaller parts namely, Migration Planning and Migration Update. The descriptions of these protocols are given below:

### 3.2.1 Migration Planning

Each tank gathers the number of friend and foe tanks in its own PE as well as all surrounding PE's within their *PerceptionDepth*. This friend and foe information is calculated as the difference between the number of Red Tanks and the number of Blue Tanks, yielding the *NetTanks* in the current PE. Since the simulation is spread across many PE's on MasPar, it will be necessary to communicate to surrounding PE's in parallel to get this information. After acquiring all the *NetTank* information, an *EnemyDirection* for each tank color (in each PE) is computed. The resulting direction is based on where the friendly tanks are outnumbered worst by the foe tanks. Each PE in which the friendly tanks are not outnumbered by foe tanks within the *PerceptionRange*, will be given an *EnemyDirection* of *None*. The Migration Planning Protocol accomplishes three main tasks — namely, gather perception, compute enemy threat and find enemy direction. More details can be found in Chandrasekharan and Vemulapati (1994).

### 3.2.2 Migration Update Protocol

All tanks will all start out with a speed equal to a user specified *FullSpeed* and attempt to go in the *EnemyDirection*. If turning is required, or difficult terrain is encountered (e.g., forests, mountains or water), *TankSpeed* will be decreased. Tanks may increase *TankSpeed* again when leaving difficult terrain, but *TankSpeed* decreased due to turning will not be gained back until the next migration cycle. Two tanks are not allowed to occupy the same level-0 point. The migration update involves the following steps.

1. Set initial tank direction: Tank information is kept in a packed list. Each PE contains a *RedTankInfo* and a *BlueTankInfo* list. In addition to these lists, a *LocalRedTankInfo* and a *LocalBlueTankInfo* list are used to store information about each tank during the migration cycle. Some of the variables in these lists, and all of the variables in the local lists, must be initialized at the start of a migration cycle.

2. Iterate through all tanks in each PE: The steps listed below will be done for every tank present in the *RedTankInfo* or *BlueTankInfo* lists.

3. Filter out all tanks that are dead, reloading, or have no moves left.

4.Check For Collision Avoidance — Given a tank's current position and direction, check if a collision will result if the tank moves in that direction. Collisions can result from encountering a terrain type of *Mountain* or *Water*. It is assumed that while tanks can not occupy the same level-0 point, they can maneuver around tanks in level-0 points without causing collisions. If a collision results, the tank will randomly choose a direction and continue to rotate in that direction (while adjusting its *LocalTankInfo.MaxTankSpeed* accordingly), until it clears the object, or reduces its *LocalTankInfo.MaxTankSpeed* to *Halt*. If a tank is attempting to move out of a PE, it may be necessary to communicate to other PE's to get information about the terrain. This is done by simply using the **Connect** and **Router** constructs in MPL.

5. Examine the level-0 points in current direction until a collision happens or if no moves are left.

6. Move the tank. Note that since tanks are being moved in parallel, it is possible

7. Clean Up — There are chances that a move may become blocked, either through other tanks occupying desired level-0 points, or, *RedTankInfo* and/or *BlueTankInfo* list becoming full. When such blocking occurs, the tank's original state is resumed and the migration is postponed until the next cycle of the migration protocol.

## 4 EXPERIMENTS AND RESULTS

### 4.1 Source Code

MasPaWS source code consists of a total of 17 program files, a data file and a Makefile. The programs have a total of, roughly, 3,700 lines code split into over 30 functions. Apart from these, there is also a C

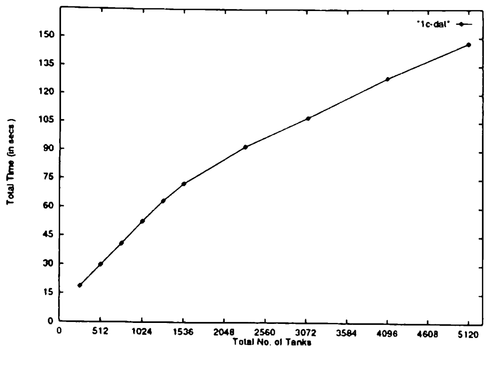

Figure 1: Number of Tanks vs Simulation Time

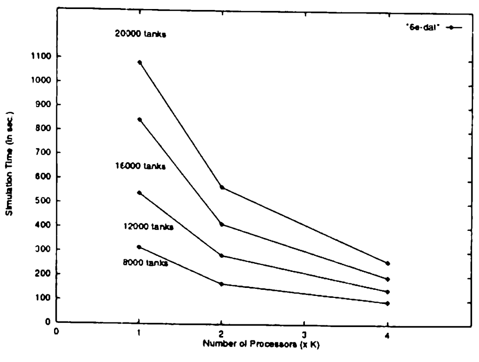

Figure 2: No. of PEs vs Simulation Time (MP-2)

program (called MAKEDATA) which automatically generates the data interactively with the user, for the simulator to act upon.

## 4.2 Impact of Number of Combat Vehicles

Figure 1 illustrates the performance of MasPaWS for increasing overall computational load, namely, the number of tanks per army, keeping UnitSize invariant at 4. The data in Figure 1 conforms to PE mesh sizes of 64 × 64 on MP-2 for 100 simulation cycles. The numbers of tanks cited are for each army.
**Comments:** MasPaWS has a linear performance characteristic against a computational load which is potentially quadratic. The robustness of the performance stands even at 5000 tanks for each side.

The plots in Figure 2 show the impact of using more processors for the same number of tanks. There are 4 plots, each run with PE mesh configurations of $32 \times 32, 32 \times 64$ and $64 \times 64$ on MP-2. The total number of tanks were varied in the range 8,000, 12,000, 16,000 and 20,000 with a view to test MasPaWS performance at high-end. Timings indicated are for 100 simulation cycles. The figure 3 shows the speedup obtained on these data which is roughly linear pointing to an efficiency of over 85%. The reader may also note the superlinear speedup for plots involving 20,000 and 16,000 tanks. Upon closer examination, we found that this was due to the probabilistic variation of the runtime of MasPaWS protocols when the same number of tanks were distributed on different mesh sizes.
**Scalability:** This is a key concept in parallel processing which describe s the performance changes as we vary the experiments. We introduce two notions of scalability which are somewhat related to each other. We call one *Data Scalability* and the other *Architecture Scalability*. Data scalability refers to propor-

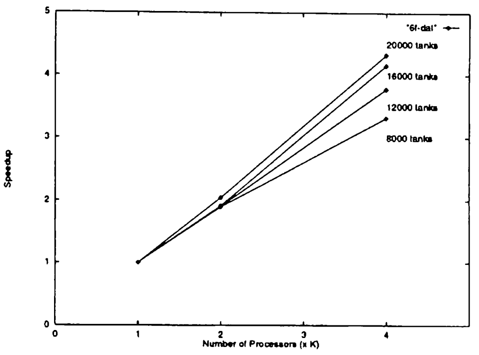

Figure 3: No. of PEs vs Speedup

tional improvement in performance as data size is increased keeping the number of processors an invariant. Architecture scalability refers to proportional improvement in performance as machine size (number of processors) is increased for the same data size. Existing parallel simulators for battlefield are known to suffer on both counts. But our experiments strongly suggest that MasPaWS is both data and architecture scalable for war simulation within the features we have considered. It does come as a surprise when many believe complex simulation is not suited for synchronous, SIMD parallelism. In the coming months, we hope to continue our study on this aspect more closely.

## 4.3 UnitSize Versus Simulation Time

Using a total of 1600 tanks distributed on 13 rows of the PE array of size 32 × 32, we varied the UnitSize (the maximum number of tanks of either color that a level-1 grid can have) as $4, 8, 16, \ldots, 50$. The UnitSize is indicative of how crowded a level-1 grid could become as the battle progresses. Hence it is a

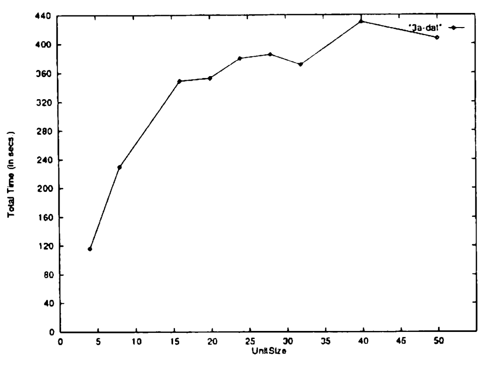

Figure 4: Simulation Time vs Unit Size

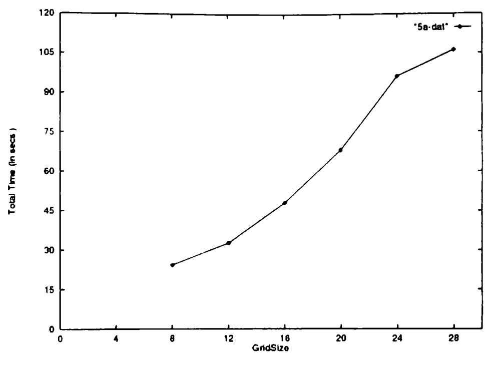

Figure 6: Simulation Time vs Grid Size

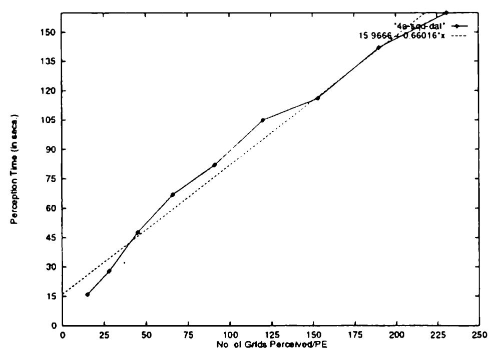

Figure 5: Simulation Time vs No. of Tanks Perceived

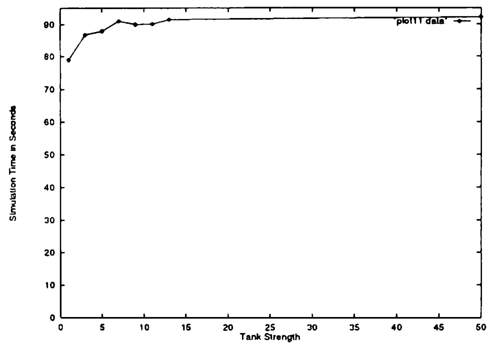

Figure 7: Simulation Time vs Tank Strength

dynamic load indicator and the Figure 4 shows the effect. The plot is linear up to a certain point and flattens out at the top. This is due to a combination of tanks spreading out as battle progresses and tanks being lost in combat.

### 4.4 Perception Complexity

The perception protocol allows for each attacking tank in a current level-1 grid to potentially engage a target tank in a neighboring grid. The perception depth $d$ is the parameter that controls how deep the attacking tank will engage. The number of neighboring grids potentially engaged by an attacking tank is given by the expression $G = (d + 1)(2d + 1)$ which are the number of grids in the facing direction of the tank. Again in the worst-case scenario, an attacking tank could examine $O(G * \text{UnitSize})$ target tanks. We suspect that this worst-case scenario happens infrequently. Keeping the UnitSize invariant we varied $d$ from 2 to 10 and plotted the number of potential Grids Perceived $G$ versus the Perception Time. This is shown in Figure 5, wherein we have plotted a possible linear-fit.

### 4.5 Effect of GridSize

GridSize, as we know, is a measure of the size of the terrain handed out to each processor at the start of simulation. In fact, the size of the terrain grows as the square of the GridSize. Terrain size contributes significantly to perception and migration complexity and hence to overall simulation time. In Figure 6, we have plotted GridSize versus simulation time on a $32 \times 32$ mesh, keeping the total number of tanks at 1024 with UnitSize of 4. In order that FullTankSpeed not have any role in effecting early or late combat (because of changes in terrain size), we set FullTankSpeed=GridSize. The plot in Figure 6 bears out the quadratic growth rate in battle complexity.

### 4.6 Strength of Tanks

In MasPaWS, the number of *hits* that is needed to destroy a tank is parametrized by TankStrength. Clearly, the only protocol that would be influenced by this parameter is the Combat protocol. During the combat phase, if a tank is fired upon by many tanks, the cumulative total needs to found at the end of the phase, to assess the damage. This is relatively time consuming on MasPar since the *hit counts* from potentially different PEs have to be added, which results in a communication overhead. When the TankStrength is at its lowest value 1, the communication overhead is minimal. As the TankStrength increases, the communication time increases also but the tanks are becoming more and more invincible. So after the initial increase , the communication time flattens out (refer to figure 7).

The various effects of varying Assets, GunRange, TankSpeed, and Terrain on the simulation time have also been studied and can be found in Chandrasekharan and Vemulapati (1994).

## 5 CONCLUSIONS

We presented the design, development and performance of a parallel war simulator to exploit SIMD architecture on several thousands of processors. From our detailed study we have come to understand that SIMD parallelism is well suited for complex and irregular simulation while achieving high efficiency. The second generation prototype of MasPaWS will be used to test multiple weapon types and battle strategies.

## ACKNOWLEDGMENTS

The first author would like to acknowledge the help of Dr. Muralidhar Medidi for valuable discussions and Dr. David Nicol for his useful e-mail replies. This work is funded by Science Applications International Corporation (SAIC) under contract number #08-940086-92.

## REFERENCES

Chandrasekharan, N., and Udaya B. Vemulapati. “Massively Parallel War Simulator (MasPaWS)”, Tech. Report CS-TR-94-01, Dept. of Computer Science, University of Central Florida, 1994.

Deo, N., M. Medidi, and S. Prasad. “Processor Allocation in Parallel Battlefield Simulation”, Proc. of the Winter Simulation Conference, pp. 718-725, 1992.

Fujimoto, R.M. “Parallel Discrete Event Simulation”, Communications of the ACM, vol. 33, No. 10, pp.31-53, 1990.

Gilmer, J.B. “Parallel Combat Simulation Research”, Tech. Report. BDM/POS-88-341, BDM Corporation, February 1988.

Gilmer, J.B., and Hong. “Replicated State Space Approach for Parallel Simulation”, Proc. of the Winter Simulation Conference, pp. 430-433, 1986.

Hawkins, T.C., and F. Thompson. “Quickscreen”, Proc. of the Winter Simulation Conference, pp. 575-585, 1985.

Nicol, D.M. “Performance Issues for Distributed Battlefield Simulations”, Proc. of Winter Simulation Conference, pp. 624-628, 1987.

Nicol, D.M. “Mapping a Battlefield Simulation onto Message-Passing Parallel Architectures”, Proc. of the SCS Multiconference on Distributed Simulation, pp. 141-146, 1988.

Nicol, D.M., and R. Fujimoto, “Parallel Simulation Today”, Manuscript, Department of Computer Science, College of William & Mary, 1994.

Prasad, S.K. “Efficient Parallel Algorithms and Data Structures for Discrete-Event Simulation”, Ph.D. Thesis, Department of Computer Science, University of Central Florida, December 1990.

Van Brackle, D. “GISMO - (Game for Intelligent Simulated Military Opponents)”, Tech. Report, Institute for Simulation and Training, Orlando, 1992.

Weiland, F., L. Hawley, A. Feinberg, M.D. Loreto, L. Blue, P. Reiher, B. Bechman, P. Hontalas, S. Bellenot, and D. Jefferson. “Distributed combat Simulation and Time-Warp: the Model and its Performance”, Proc. of the SCS Multiconference on Distributed Simulation, pp. 14-20, 1989.

## AUTHOR BIOGRAPHIES

**N. CHANDRASEKHARAN** is an Assistant Professor in the Department of Mathematical Sciences at the Loyola University of Chicago. His principal areas of research interest are in parallel algorithms, parallel simulation, and combinatorial optimization. He is a member of ACM and IEEE Computer Society.

**UDAYA B. VEMULAPATI** is an Assistant Professor in the Department of Computer Science at the University of Central Florida, Orlando, His research interests are in the parallel scientific computing, parallel simulation, parallel and distributed operating systems. He is a member of ACM and SIAM.

**ALLEN T. IRWIN** is a vice president and senior engineer with Science Applications International Corporation (SAIC) in Orlando, FL. His chief areas of interest are simulations and training devices. He is a member of ACM and the IEEE Computer Society.