Analytic Performance Models Of Parallel Battlefield Simulation Using Conservative Processor Synchronization

THESIS

James B. Hiller
Captain, USAF

AFIT/GCS/ENG/94D-08

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/94D-08

Analytic Performance Models Of Parallel Battlefield Simulation Using Conservative Processor Synchronization

THESIS

James B. Hiller
Captain, USAF

AFIT/GCS/ENG/94D-08

AFIT/GCS/ENG/94D-08

Analytic Performance Models Of Parallel Battlefield Simulation Using Conservative Processor Synchronization

THESIS

Presented to the Faculty of the Graduate School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science

James B. Hiller, B.S.C.S
Captain, USAF

December 1994

Acknowledgements

My first gratitude goes to Dr Hartrum. His patience, pragmatism, and fairness guided me through this thesis and hopefully rubbed off on me as I untangled the webs of confusion. My thesis committee, Maj Luginbuhl and Dr Santos, were most supportive of the ideas I conjured up in this work. And, while Dr Lamont wasn't on my committee, his guidance through CSCE 786 and *Gödel, Escher, and Bach* fostered ideas and awareness that undoubtedly influenced my thesis work and everything else I'll do.

I owe a sincere thanks to the latest BATTLESIM developers, Ken Bergman and Wally Trachsel. Their hard work gave me a parallel battlefield simulation that worked extremely well. They'll probably never see this, but I want the record to show that there has been at least one GCS thesis at AFIT that wasn't bogged down in code repair.

I'm also indebted to a number of people who graciously extended their patience and friendship during the hundreds of nights at work. Around the clock and just about every day, George Gates, Jay Graham, and Mike Gudaitis were always willing to listen and help when they could. But most of all, I owe everything to my wife and daughter, Pam and Shana. It was only with their quiet love and strength from 1300 miles away that I was able to learn as much as I did. Their sacrifice for the last nineteen months was far greater than mine. I hope I'm worthy of it. Thank you both for keeping Texas under control without me.

James B. Hiller

Table of Contents

	Page
Acknowledgements	ii
List of Figures	iv
List of Tables	v
Abstract	vi
I. Introduction	1
1.1 Background	1
1.1.1 Computer Simulation	1
1.1.2 Department of Defense (DoD) Applications of Simulation Technology	2
1.1.3 Battlefield Simulation and Parallel Computation Technologies	3
1.1.4 Performance Modeling for Insight	4
1.2 Problem Statement	5
1.3 Research Objectives	5
1.4 Scope and Exclusions	6
1.4.1 Model Development and Experimentation	7
1.4.2 Other Excluded Areas	8
1.5 Desired End Results	9
1.6 Methodology	9
1.6.1 Literature Review	9
1.6.2 Model Development and Validation	9
1.7 Structure of Thesis	10

Page

II. Background and Literature Review 11

2.1 Simulation Architecture and Design 11

2.1.1 Control Flow 11

2.1.2 Major DES Software Components 12

2.1.3 Typical Architecture 13

2.1.4 Software Engineering Principles 14

2.2 Problem Decomposition 15

2.2.1 Proximity Detection 15

2.2.2 Decomposition in PDES 16

2.3 Processor Synchronization in PDES Execution 18

2.3.1 Logical Process (LP) Distributed Computation Model . 18

2.3.2 Synchronization Strategies 19

2.4 Anatomy of PDES 20

2.5 Performance Modeling and Analysis 20

2.5.1 Model Usage 21

2.5.2 Model Development Considerations 22

2.6 Software Engineering Principles Revisited 23

2.7 Summary 24

III. Design Study and Definition Framework 26

3.1 Overview 26

3.2 Simulation Design and Limitations 26

3.2.1 Simulation Architecture 26

3.2.2 BATTLESIM Application Layer and OO Design 27

3.2.3 Design Limitations 32

3.3 Definition Framework 35

3.3.1 Notation and conventions 35

3.3.2 Definition Framework 37

3.4 Summary 40

Page

IV. Performance Model Derivation 41

4.1 Performance Model Definitions and Descriptions 42

4.1.1 Cardinality and Arithmetic Definitions 42

4.1.2 Reference Conventions 42

4.1.3 Abstract Data Types and Function Definitions 43

4.1.4 Operation Definitions 44

4.2 Sequential Simulation 45

4.2.1 Canonical Sequential Simulation Algorithm 45

4.2.2 Sequential Performance Model 45

4.3 Parallel Simulation 54

4.3.1 Canonical Parallel Simulation Algorithm 54

4.3.2 Input Protocol Model 58

4.3.3 Output Protocol Model 67

4.3.4 Integrated Parallel Performance Model 70

4.4 Summary 70

V. Model Analysis and Demonstration 72

5.1 Performance Models and Reduction 72

5.1.1 Assumption Set 1 76

5.1.2 Assumption Set 2 78

5.1.3 Parallel Overhead Model Reduction 81

5.1.4 Algorithm and Performance Model Review 83

5.2 Measurements and Demonstration of Model Results 87

5.2.1 Basic Operation Timing 87

5.2.2 Simulation Experiments 92

5.3 Analysis 94

5.3.1 Parallel Performance Issues 94

5.3.2 General Performance Issues 105

5.4 Summary 108

Page

VI. Summary of Results and Recommendations 109

6.1 Summary of Results 109

6.2 Research Contribution 110

6.3 Recommendations for Further Study 111

6.3.1 General Directions 111

6.3.2 Specific Issues 112

6.4 Summary 114

Appendix A. Data Dictionary and Function Definitions 115

A.1 Data Dictionary 115

A.1.1 Definitions and Descriptions 115

A.1.2 Structural Constraints 121

A.1.3 Data Model Interpretation 121

A.2 Performance Model Definitions and Descriptions 122

A.2.1 Cardinality and Arithmetic Definitions 122

A.2.2 Reference Conventions 123

A.2.3 Abstract Data Types and Function Definitions 124

A.2.4 Operation Definitions 124

A.3 Summary 125

Appendix B. The General Task Allocation and Scheduling Problem 126

B.1 Task Allocation and Scheduling Approaches 126

B.1.1 Nearest-Neighbor (NN) and Recursive Clustering (RC) 127

B.1.2 Heavy Node First (HNF) and Weighted Length (WL) . 128

B.2 Summary 130

Appendix C. AFIT PDES Research 131

C.1 AFIT PDES Testbed 131

C.2 Object-Oriented Parallel Simulation 133

C.3 Parallel Simulation Task Scheduling Research 134

Page

Bibliography 136

Vita 141

List of Figures

Figure | Page
1. General DES Architecture 13
2. Simulation Architecture with Components 21
3. BATTLESIM, TCHSIM, and SPECTRUM 28
4. Canonical Sequential Algorithm 46
5. Canonical Parallel Algorithm - Initial Step 55
6. Canonical Parallel Algorithm 56
7. Completely Independent Event Schedules 99
8. Dependent Event Schedules 99

List of Tables

Table Page

1. BATTLESIM Operation Time Measurements 88
2. Collision Prediction Times 90
3. Individual Collision Execution Times (in msec) 91
4. NEQ Operation Measurements 92
5. Synchronization Algorithm Improvement Results 93

AFIT/GCS/ENG/94D-08

Abstract

This study investigated the development and use of analytic models for performance analysis of parallel discrete event battlefield simulation using conservative synchronization. A simulation architecture with layered application, simulation, and host machine services provided the model development basis. Simulation entities were modeled with set-theoretic definitions. Deterministic performance models using these definitions were developed for event prediction, scheduling, and execution in sequential battlefield simulation. The sequential model was expanded to include relative bounds for overhead factors introduced when the simulation is spatially decomposed for a parallel distributed memory machine. Comparison of sequential and parallel models instantiated for a simulation with uniform workload showed a potential for unbounded processor blocking. A synchronization algorithm modification to limit per-iteration blocking is shown theoretically to decrease finishing time. Modification results were demonstrated on a hypercube architecture. Demonstration showed that a sequential simulation requiring 60 seconds to run was limited to a best time of 30 seconds on four processors without algorithm modification. The time was improved to 17 seconds using the modification. A number of basic timing measurements also showed that event list operations on a sequential structure take significantly longer than interactive event prediction algorithms using simulation entities maintained in similar structures.

Analytic Performance Models Of Parallel Battlefield Simulation Using Conservative Processor Synchronization

I. Introduction

Large-scale simulation requirements are often accompanied by a need for speed. Sequential simulations may be insufficient for modeling complex processes with large data requirements or irregular periods of intense activity over a long time interval. Detailed combat simulations involving campaign activities such as logistics operations, complex air engagement patterns, electronic warfare operations, and ground attack present such challenges. Parallel implementation provides an opportunity to capitalize on advanced architectures.

The goal of this research is to model the performance of spatially decomposed parallel discrete event battlefield simulation using conservative processor synchronization. Analytic performance models reflecting the relationships between structural algorithm components are developed as the primary vehicle for the investigation.

1.1 Background

1.1.1 Computer Simulation. Computer simulation allows flexible experimentation with real-world entity and process models. Simulation is often useful when the simulated system is too expensive, time-consuming, or deadly to observe directly. Simulation also permits abstraction of details present in the real system that may obscure understanding of the system at a higher level. Used as a training aid, simulation provides a mock-up of the real system that may not otherwise be effectively replicated without undue expense or exposure to operational hazards. As a decision support tool, computer simulation can reduce risks of failure. Simulations can also be used in system development to assess design and implementation decisions before a real commitment is made.

Computer simulation benefits from the theoretical underpinnings of discrete mathematics, computation theory, and areas of mathematics pertinent to the simulated system. A simulation can be run in a manner consistent with prudent experimentation principles to explore statistical hypotheses about the simulated system. The simulation itself can be subjected to analysis and reasoning with an eye towards improving its performance or representative quality.

1.1.2 Department of Defense (DoD) Applications of Simulation Technology. The DoD exploits simulation technology in many application areas, including cockpit training and development, force employment strategy, communication system modeling, Very Large Scale Integrated (VLSI) circuit design, medical diagnosis and procedures, and airframe development. *Simulation and modeling* has been identified by the DoD as one of twenty technologies "critical to ensuring the long-term qualitative superiority of United States weapon systems" (42:519). "Simulation and modeling offer an affordable alternative to extensively testing hardware, to training with actual systems, and to developing new battle tactics and force employment concepts" (42:519). Battlefield simulation is an example of the last of these areas and is the application of interest for this research.

Simulation architecture design has become a pressing issue within DoD in recent years. The need for multi-service, interoperable simulation and modeling capabilities has given rise to the Joint Modeling and Simulation System (J-MASS) and related development projects. The thrust of these efforts is to develop platform-independent simulation systems for sequential and distributed environments, capable of supporting rapid model component exchange.

Realization of this goal offers the potential of a single, joint simulation architecture to provide flexible support while saving costs associated with software redesign. However, it also requires modular designs that separate application and simulation implementation details. A better understanding of the balance between application specifics and performance expectations is needed to construct architectures that can be used to meet cost, performance, and usability requirements.

1.1.3 Battlefield Simulation and Parallel Computation Technologies. Battlefield simulation is an inherently intensive computational task. With current and projected DoD fiscal restructuring and planned long-term reliance on simulation as a cost-saving strategy, combat simulation size and complexity is likely to grow over time. Parallel computing technologies hold the most promise for meeting future battlefield simulation performance demands. "The most significant improvements in performance now come from widespread exploitation of parallelism, and only secondarily from faster circuit technologies" (18:213).

Discrete event simulation (DES) is an event-driven simulation approach in which the simulation changes state upon event execution. This technique differs from time-driven methods in which time and simulation state are updated at uniform intervals, regardless of whether the resulting state change is of interest to the observer. Though suitable for modeling synchronous systems and processes, the performance of time-driven techniques suffers in applications in which the time intervals between observable events vary widely. DES is typically the best approach for combat and force employment simulation models (18:221).

Parallel discrete event simulation (PDES) is normally viewed as the decomposition of a single DES program into logical units that can be executed concurrently on distributed processing architectures (18:214). Many of the difficult problems associated with parallel algorithm design affect DES decomposition and concurrent execution: optimal resource and task scheduling, data consistency, workload balance, processor synchronization, and design tradeoffs against performance. Many challenges are related to the combinatoric complexity of finding the best approach for a given problem.

Synchronous behavior in some parallel algorithms allows for somewhat more reliable performance optimization techniques. As an asynchronous approach, PDES tends to be a difficult parallel algorithm design problem. Moving object PDES, exemplified by battlefield simulation, further complicates matters with highly irregular structures and data dependencies that vary unpredictably in space and time.

1.1.4 Performance Modeling for Insight.

Performance modeling is a useful step in understanding relationships between algorithm components and data dependencies that affect performance. Performance models provide abstract algorithm representations as a basis for formal reasoning about and exploration of the relationships between algorithm components. When a simulation is the subject of study, analytic modeling techniques can provide insight useful for improving the simulation and its ability to support operational requirements.

The published research dealing with PDES design issues has often focused on measured performance or analytic study using queueing system and circuit models. These approaches can provide very good results and when applied to PDES systems modeling environments with similar behavior. However, many of the dynamic aspects in combat models do not resemble those in other problem models. Treatment of combat simulation in the literature has focused on empirical studies using application-specific simulation environments (53) (18:218) (12).

1.2 Problem Statement

The goal of this research is to develop analytic performance models of sequential and parallel discrete event battlefield simulation using spatial problem decomposition, a layered simulation architecture, and conservative parallel synchronization. The models are intended to support identification of factors that determine parallel performance.

1.3 Research Objectives

Sequential algorithm performance models provide a comparative foundation to determine the effectiveness of decomposition strategies prior to design and implementation. Sequential performance models can be expanded and manipulated to postulate data structure and organization impacts. Parallel performance models provide a means for evaluating and improving data distribution and process synchronization methods. Along these lines, several issues are analyzed for simulations exhibiting predictable behavior.

1. **Decomposition granularity.** Research activities to date have not identified heuristic methods, break-even points, or cost-benefit tradeoffs for spatial decomposition in

moving object simulations (53). Methods for estimating decomposition costs are proposed.

2. **Conservative processor synchronization.** The usefulness of conservative synchronization protocols for parallel battlefield simulations has been challenged based on the dynamic character of combat environments (36:515) (4:87). No formal analysis has been offered to support such challenges. This effort includes an initial examination of a general conservative algorithm with deadlock prevention. The examination is based on algorithm structure analysis guided by the performance models developed.

The underlying goal is to support efficient production of decomposition and synchronization alternatives that offer predictable performance with minimum reliance on application knowledge or simulation architecture.

1.4 Scope and Exclusions

The ability to achieve revolutionary results is determined by the existence of a significant framework of validated theoretical work that can be analyzed and applied to the current problem. The literature provides such a framework, both analytically and empirically, for general parallel algorithms with regular structure and conservative processor synchronization algorithms for certain types of simulations. However, many of the premises behind the results reported do not hold for battlefield simulations using spatial decomposition.

The intent of this effort is to lay a theoretical foundation upon which future work can advance. Accordingly, certain considerations scope performance model development and experimentation.

1.4.1 Model Development and Experimentation. Performance models are based on abstractions of battlefield simulation algorithms used at the Air Force Institute of Technology (AFIT). The resulting models represent general battlefield simulation solutions to the extent that simulation design approaches used at AFIT represent approaches used elsewhere.

Previous research at AFIT has focused on design issues associated with parallelization of sequential battlefield simulations. Performance model development in this work is an initial effort to formally capture basic performance elements. Impacts introduced by object-oriented (OO) implementation, advanced application feature modeling, alternative decomposition strategies, and different synchronization protocols are not addressed.

Probabilistic performance models are useful when random variables can be associated with known probability distributions. Without such knowledge, performance models can be derived but may be of little use in studying behavior. Since descriptions of dynamic behavior for battlefield entities are not available, the models capture predictable, static algorithm structure rather than dynamic application behavior.

No attempt is made to improve the representative or functional quality of the BATTLESIM validation platform. Flaws discovered in BATTLESIM are resolved to the extent necessary to complete demonstrative experiments and measurements. Battlefield behavior implementation is comparable to that used in similar studies (for example, (12:719) (29:143)).

The effects of significant performance parameters that are intrinsically dynamic for general battlefield simulations are analyzed informally. Performance models capture these parameters to some extent but do not address the complex effects of dynamic parameters allowed to vary over their entire domain. This is a refinement left open for examination should probability functions for battlefield entity behavior become available. Reduction is used to demonstrate algorithm behavior of these parameters over constrained limits of measurability.

The deadlock avoidance algorithm described by Chandy and Misra (7) is the basis for synchronization protocol study. The algorithm is described in detail in Chapter IV.

Processor utilization on the validation system is limited to one process per physical processor. Workload distribution among processors is dynamic based on simulation object movement in the battlefield space as mapped to physical processors during compilation.

System measurement and result demonstration is conducted using the Intel iPSC/2. The machine is a hypercube architecture, representative of current distributed memory machines.

1.4.2 Other Excluded Areas.

1.4.2.1 Operating System Interface. The system-level component of the simulation architecture supporting BATTLESIM is the Simulation Protocol Evaluation Testbed using Reusable Models (SPECTRUM) (34). SPECTRUM provides the sole interface to operating system communication services for BATTLESIM. SPECTRUM also implements the parallel processor synchronization protocol. The current implementation of SPECTRUM is used generically for a variety of simulation tools (21).

1.4.2.2 PDES Input and Output. PDES interaction with mass storage and other peripherals has been shown to degrade parallel implementation performance benefits (6:80). No special support for parallel mass storage is used. Since BATTLESIM does not use mass storage after initialization, performance models and demonstration do not extend in that direction.

1.5 Desired End Results

The intended result is a set of performance models that can be used to study the performance impact of design changes to battlefield PDES strategies using conservative processor synchronization with deadlock prevention.

1.6 Methodology

1.6.1 Literature Review. The literature review surveyed recent work and results pertaining to general parallel algorithms; PDES using conservative processor synchronization; and analytic modeling techniques. Special emphasis areas included problem decomposition methods, predictive performance modeling, and results for parallel battlefield simulations.

1.6.2 Model Development and Validation. Performance model development was based on a framework of set-theoretic simulation structure definitions. Canonical simulation algorithms operating on data model entities and relationships were developed as a starting point for analytic performance model derivation. Modeling assumptions and reduction results were demonstrated using battlefield simulations developed for a distributed memory parallel machine.

1.7 Structure of Thesis

This chapter describes the motivation, objectives, and approach for this thesis. Chapter II summarizes relevant parallel discrete event simulation literature. Chapter III describes the design of the simulation environment used as the basis for performance analysis and modeling and delineates a basic set of data definitions used for performance model construction. Chapter IV develops an expanded set of definitions used explicitly in performance modeling, canonical sequential and parallel battlefield simulation algorithms, and resulting performance models. Chapter V details performance model derivations, test descriptions and data associated with measurement and demonstration, and result analysis with respect to battlefield simulation algorithm design. Chapter VI concludes the thesis with a summary of results and recommendations for further research.

II. Background and Literature Review

The problem addressed in this investigation involves concepts and issues spanning several different research areas, including:

- Simulation architectures
- Problem decomposition and parallel algorithm design
- Processor synchronization
- Object-oriented software design
- Performance modeling

This chapter presents background and recent research results that frame this investigation. The survey includes journal and conference articles published over the last seven years.

2.1 Simulation Architecture and Design

Designing an application to exploit parallelism requires an understanding of its inherent structure. Several basic aspects characterize most DES designs. Design choices are predicated on intended simulation use and include consideration of control flow strategy, software design, and overall resource requirements. This section reviews the general DES architecture commonly used with respect to problems and issues in parallel algorithm design.

2.1.1 Control Flow. An *event-driven* policy models asynchronous behavior in the physical system. State updates occur in simulation time only as they occur in real time, dispensing with computation at intervals between event occurrences in the physical system. Time is used passively to order event execution and state update. In this capacity, time ordering preserves *causality* relationships among physical system entities and processes.

The event-driven method is used to simulate environments in which time intervals between observations of interest vary widely during the simulation (17:19). Battlefield simulations fit into this category. Combat activity takes place at different intensities depending on operational tempo. Intensive activity in combat engagements occurs in minutes, while supply operations may run for months. Thus, *discrete event simulation (DES)* is the natural subject for this investigation.

2.1.2 Major DES Software Components. Typical DES designs incorporate several data components to generate and organize events.

- **Simulation Clock.** The clock is a passive entity that tracks simulation progress. The clock provides a basis for future event generation and ordering.
- **Next Event Queue (NEQ).** The NEQ is a time-ordered priority queue that holds scheduled events awaiting processing. The event at the head of the queue is the *next event* for the simulation.
- **State Variables.** State variables model entities in the simulated system. State variable values change in accordance with the behavior model representing processes in the simulated system and reflect its current state.

Major processing activities include event generation, scheduling, retrieval, and processing. Event generation is accomplished in software by implementation of a computational model that reflects the causal requirements for and results of simulated processes acting on simulated entities. The implementation uses current state variable values to compute events to be scheduled. A scheduler manages event placement in the NEQ. An event dispatcher retrieves the *next event* from the queue to be processed in accordance with the behavior model.

Many design and implementation approaches exist for DES data components and processing activities. Requirements for simulation correctness, performance, flexibility, and maintenance determine which options are most appropriate for a given simulation need.

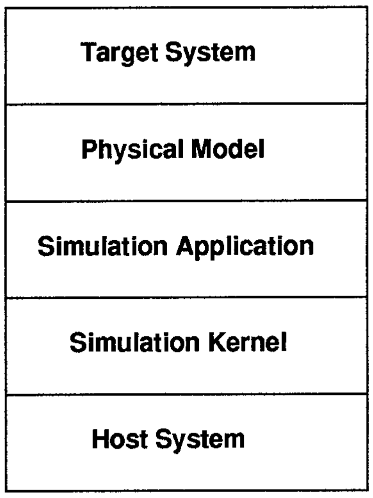

Figure 1. General DES Architecture
(32:52)

2.1.3 Typical Architecture. A typical layered DES architecture can be represented as shown in Figure 1. The *target system* is the system being simulated. The *physical model* represents the target system that models simulated entities, relationships among them, and behaviors associated with entities, along with conditions and constraints that define a context in which relationships hold and behaviors occur.

The *simulation application* represents the physical model (32:52) as a computational model in which state variables capture the observable simulated entity states and algorithmic representations effect state changes in accordance with the relationships, conditions, and constraints found in the physical model. The application includes algorithms for event generation and execution. The *simulation kernel* contains data and algorithm components that form the abstract machine used to interpret the simulation application (32:52). These include the NEQ, clock, event scheduler, and dispatcher. Finally, the *host system* is the physical machine and set of services that interprets the simulation kernel.

2.1.4 Software Engineering Principles. Within and between layers, many software engineering maxims and OO design principles are useful in DES design. In the application layer, simulated entities and behaviors are captured as objects and methods, permitting modular replacement and design component reuse. Abstract relationships can be implemented as decentralized management objects under application control. Similar approaches apply to abstract data types and operations in the simulation kernel. Modular interfaces between layers enforce loose coupling to facilitate portability and reuse among layer designs and implementations.

OO design issues and tradeoffs such as instance management and object communication overhead pose challenges in simulation as well as in other applications. Nonetheless, OO simulation offers enough benefit to encourage its use (38:280). The benefits realized from these principles are especially important with respect to requirements for flexible, interoperable simulation architectures for DoD applications.

2.2 Problem Decomposition

Problem division for allocation to available processors is one of the first areas of interest in parallel algorithm design. The goal is to reduce processor idle time while maintaining correct behavior in spite of distributed execution. The *proximity detection* problem serves to show the potential effect of decomposition on performance, whether for sequential or parallel moving object DES.

2.2.1 Proximity Detection. Event prediction in a DES requires efficient algorithms for simulating autonomous and interactive object behavior. Object locations and distances from other objects determine the opportunity for both types of behavior. Distance and event calculations involve object location and velocity data. As object attributes, such information should be hidden within each object rather than centrally managed (39:249). Centralized position data management in sequential simulations can be extremely expensive since every event computation for an object requires reference to the data of every other object in the simulation.

Battlefield sectoring has been proposed as a way to restrict the number of players that must be considered during event prediction (53:916). Sector boundaries are added to the battlefield representation and boundary crossing events are added to the behaviors modeled by the simulation. Before two objects in neighboring sectors can interact, one must cross into the other sector. Sectoring allows event prediction for a given object to be limited to consideration of objects in the same sector.

Sector boundaries serve as an implicit proximity-based indexing structure that allows computable, direct reference to object groups. In effect, sectoring provides a finer indexing granularity for data structures holding object state data. Boundary crossing events and their associated event handlers provide a clean, dynamically computable mechanism within an OO DES design by which to update the player library within any particular indexed sector area.

2.2.2 Decomposition in PDES. When decomposing a spatially oriented simulation for parallelism, sectoring provides an immediate way to divide the group of objects into easily represented subgroups for allocation to different processors. Data allocation in this fashion is referred to as Single Program Multiple Data (SPMD) decomposition, a method well-suited for distributed machine architectures (14:8).

SPMD decomposition is a strategy for localizing dependent data items on each processor (14:8). For application to battlefield PDES, groups of sectors are mapped to processors in the parallel architecture. Objects move between sectors, and thus processors, based on their changing location values. Decomposition also distributes the NEQ and the event generation, scheduling, and execution mechanisms.

Spatial decomposition is a reasonable approach for moving object PDES. Since object interaction is limited by distance, state variable references among several objects can often be confined to a single processor. However, object features such as sensor range or physical size can result in the need to share object attribute values across several processors. A data replication scheme must be used to ensure that all shared state variables among processors are consistent with respect to simulation time (17:21).

Object movement across sector boundaries requires object remapping from one sector to another to preserve the relationship between object locations and sector definitions. Similarly, objects must be transferred among processors when crossing sector boundaries coincident with processor boundaries. Processor synchronization is required to ensure correct processor state update with respect to time (17:21). Synchronization tends to be a key problem in PDES design.

Decomposition techniques include both *static* and *dynamic* approaches. Static approaches capitalize on regularities in problem structure and data representations to achieve reasonable parallel performance without incurring dynamic reallocation overhead. Dynamic, or *adaptive*, approaches shift workload after initial static decomposition so as to maintain processor efficiency in the face of changing data or task dependencies.

Static approaches used alone virtually guarantee an imbalanced workload among processors (32:52). Processor workload is dependent on both the number of objects on the processor and the number and complexity of events processed for each object over a given time interval. Workload imbalance affects the overhead attributable to synchronization and decreases exploited problem parallelism by forcing some processors to be idle while others have considerable work to do.

2.3 Processor Synchronization in PDES Execution

When state variables and operations on them are distributed over a processor network, computation results must be shared to cooperatively determine the next event or correctly update state variables. Distributed algorithms must have provisions to send and receive data across the network and either prevent or detect and predictably recover from state inconsistency, or *causality*, errors (17:20). Synchronization actions normally cause a process to wait an arbitrary amount of time until a communication is received from another process. Processors may deadlock if cyclic dependencies exist between them. Emphasis is placed on preventing or recovering from deadlock and minimizing synchronization activities (33:61).

The most general synchronization algorithms tend to approach the performance of barrier synchronization (17:24). Research directed at optimizing synchronization normally assumes some problem and solution knowledge. Resulting performance improvements support domain knowledge usage in removing synchronization delay and increasing exploited parallelism (19:3).

Interest in algorithms that guarantee causality correctness in PDES has led to development of a computation-theoretic process model (27:51) over which formal simulation correctness proofs can be derived. The model is similar to models associated with communicating sequential processes (22).

2.3.1 Logical Process (LP) Distributed Computation Model. A logical process (LP) is a computational entity that can "execute sequential code and two special commands: *receive* and *send*" (27:51). A receive command specifies a communication channel from which to take an incoming message, while a send command specifies a channel upon which to place an outgoing message. When an LP executes a receive, there may be an arbitrarily long delay until the next message is available. When the next message arrives, the LP continues sequential execution. A message sent by an LP may take an arbitrary but finite time to arrive at its destination. In all cases, communication channels observe a first-in-first-out discipline with respect to messages carried. (27:51)

The LP model supports layered simulation PDES design by encapsulating synchronization and communication operations. Designs and implementations based on the model benefit from decoupling between the simulation kernel, the synchronization protocol, and the underlying operating system communication services. Inherent modularity imposed by the model also facilitates independent behavior analysis of the synchronization protocol and other simulation components.

2.3.2 Synchronization Strategies. Strategies for achieving cooperation between processors are known as *synchronization protocols.* Protocol development and speculation about performance properties has produced a vibrant body of research activity. Two protocol families, *conservative* and *optimistic*, have emerged in the literature. The main difference between the two families is the error handling policy enforced.

Conservative protocols prevent causality errors, while optimistic protocols detect and recover from causality errors (18:216) (17:6,17). Arguments have been offered asserting that synchronization is more aptly viewed as a continuum that ranges from conservative to optimistic (35:671). Regardless of the taxonomy, synchronization methods depend on knowledge of dependencies determined by the application and are thus tied to its dynamics.

Shifting data dependencies across processors such as those found in moving object and battlefield PDES appear to exacerbate performance problems in conservative protocols (17:21).

2.4 Anatomy of PDES

By applying the problem decomposition process to the general DES architecture and adding a synchronization mechanism, a sequential DES can be readily transformed into a PDES and mapped to a parallel machine. The host machine provides physical process entities and process management services represented abstractly by the LP model. The host also provides interprocess message primitives described abstractly in the LP model. Synchronization algorithms are logical operators that control interprocess message flow. In PDES designs based on the LP model and that resemble the general DES architecture, the message passing and synchronization components are both part of the host machine layer. Replication over multiple processors results in transformation from sequential DES to SPMD PDES.

The general simulation architecture diagram is annotated in Figure 2 to show how the various simulation components might be mapped to the architecture for a particular application and machine.

2.5 Performance Modeling and Analysis

Performance models capture relationships between parallel application components that affect execution time, throughput, or other useful metrics (45:271). In addition to

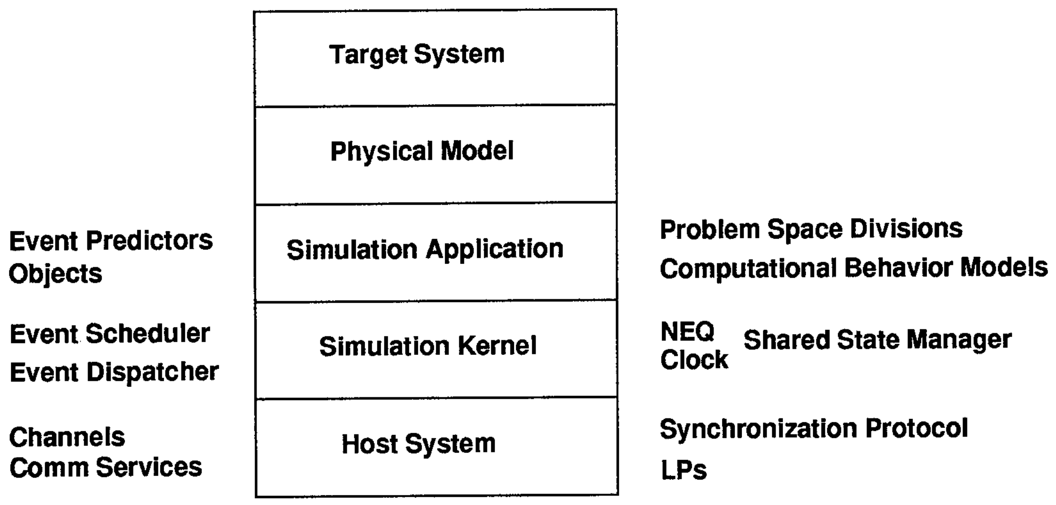

Figure 2. Simulation Architecture with Components

providing a cogent understanding of component relationships, an accurate performance model may be used to predict absolute performance resulting from particular decisions.

2.5.1 Model Usage. Performance models can be either deterministic or probabilistic. A deterministic model is one in which all variables are deterministic rather than random. A probabilistic model has at least one random variable. Deterministic performance models often sacrifice accuracy for simplicity, while probabilistic models offer the opposite tradeoff (15:123,160-5). Model parameters associated with static algorithm structure or fixed data dependence are often deterministic, while parameters associated with unknown computation results or input values are at best probabilistic. Little can be done with a probabilistic parameter if its probability functions are unknown. The accuracy and predictive abilities of a model decrease as knowledge of significant parameters decreases.

Performance model utility, regardless of type, comes from the ability to alter model parameters that correspond to configurable portions of the system being modeled. Depending on model accuracy, changes realized by altering parameters can be used to predict results from similar changes in the modeled system. In the best case, the model can serve as a descriptive function upon which a decomposition or tool can be developed. Such tools allow the application designer to project the results of particular decomposition strategies and ultimately can be used to automate the task. (54:164)

2.5.2 Model Development Considerations. Parallel algorithm performance metrics are normally based on comparisons, either modeled or measured, between parallel and sequential implementations (25:117-21). The comparison requires accurate model-based estimates or empirical measurements to produce useful results. Model accuracy depends on how well the model captures the combined effects of the algorithm, data management, operating system, and parallel hardware architecture.

Application data dependencies often determine algorithm behavior. In moving object PDES, data dependence affects sequential application algorithm performance and parallel synchronization algorithm behavior. For example, sectoring may speed up an event prediction algorithm but total cumulative savings depend on object movement. Likewise, the amount of concurrent execution achieved varies with many factors, including object and event distribution over processors and time. Complex data dependence challenges the development of usable moving object PDES performance models.

Basic system operating characteristics influence the performance of a parallel implementation. Time needed to send, receive, and process messages can be significant with respect to computation performed. Unpredictable variations in basic functions can influence model accuracy as well, but the impact from variability may be overshadowed by inefficiency in implementation (12:718). Incorrect model assumptions about the amount of variation can also lead to inaccuracy (1:61).

2.6 Software Engineering Principles Revisited

The primary factors affecting PDES performance are problem decomposition, synchronization algorithm behavior, and communication costs induced by synchronization. Improvements accrue from embedding application knowledge in the synchronization algorithm design (17:24). However, encapsulation that supports generality and reusability removes application knowledge from synchronization. The modeling task then becomes to show how much application knowledge is needed for minimum acceptable performance. The complementary design task is to develop an architecture that provides the right amount of application knowledge without compromising flexibility.

Appendix B contains a brief explanation of the general task allocation and scheduling problem for parallel algorithms. The explanation is accompanied by descriptions of several heuristic approaches recently proposed in the literature: Nearest-Neighbor, Recursive Clustering, Heavy Node First, and Weighted Length.

To allay the intractability of the general scheduling problem, each approach uses application knowledge developed statically or dynamically to demonstrate the possibility of acceptable performance using the approach. When fundamental relationships can't be determined prior to execution, static decomposition must include assumptions about precedence, as well as provisions for correction when the assumptions are violated during execution (14:8). In PDES using SPMD decomposition, data dependencies replace task dependencies, and the synchronization algorithm replaces the scheduling algorithm. Application knowledge needed to yield acceptable performance is likely to have a counterpart as well. The survey of methods in Appendix B demonstrates the types of application knowledge that have been used with varying degrees of success in parallelization efforts.

The need to make application knowledge available to the synchronization algorithm when using conservative methods has been shown empirically (17:22) (16). Since the approach works entirely with event and message timestamp information, extraction of parallelism is dependent on accuracy of future event times available to the algorithm (17:22). One inherent limitation is the fact that two sets of events may be completed unrelated to one another and thus could be executed independently regardless of their time relationship. Since the synchronization algorithm does not use event dependency information, this type of parallelism cannot be recognized.

2.7 Summary

> Military combat models differ from many traditional simulation application areas (e.g., circuit simulation) in that any of the entities may potentially interact and that these interactions are difficult to predict. The highly dynamic nature of these problems thus precludes the static mapping or determination of simulation actor interaction. Thus, in the general case, the conservative approach can not be effectively utilized in this class of problem. (36:515)

This position discounts the use of application knowledge to restrict potential interactions. Spatial restriction and conservative synchronization have been used to achieve promising processor efficiencies in time-driven battlefield simulations (29:141), (12:718). Similar techniques, combined with judicious use of application knowledge, may produce acceptable results in battlefield PDES using conservative synchronization. Recent work alludes to this possibility (16).

The performance models developed in Chapter IV provide a starting point for formal examination of the parallel performance benefits that may be associated with use of application knowledge. The next chapter contains a design description of the simulation environment used to develop the performance models.

III. Design Study and Definition Framework

3.1 Overview

This chapter describes the salient details and limitations of the simulation model used for the performance analysis. The description is followed by a design-based data definition framework that formally describes the significant data entities used in performance model development.

3.2 Simulation Design and Limitations

This section discusses the design of BATTLESIM, the AFIT environment used to investigate parallel battlefield simulations. BATTLESIM is integrated with TCHSIM, a collection of general DES mechanisms and services, and SPECTRUM, a parallel simulation protocol testbed based on the LP model. The environment is similar to many kinds of parallel simulators using coarse-grained, SPMD decomposition; spatial relationships among interacting simulated entities; an event-driven discipline; and conservative processor synchronization.

3.2.1 Simulation Architecture. BATTLESIM is a battlefield PDES application supported by several lower-level components. The underlying simulation kernel is TCHSIM, an object-oriented collection of structures and operations that provide basic simulation services. TCHSIM provides a simulation clock, a NEQ, a top-level event dispatcher, and structural definitions for simulation events and relationship mappings. The structural definitions are part of the abstract interface between the application layer and the simulation kernel. The remainder of the interface is explicit as defined by methods in TCHSIM services.

Host machine process management and communication services supporting TCHSIM are provided by the AFIT version of SPECTRUM. SPECTRUM manages the creation of and communication between each LP. In this case, the sequential code in each LP is a complete DES. Support processing within SPECTRUM includes both an input/output message handler and synchronization logic for inter-LP message flow control. Flow control logic is known in SPECTRUM nomenclature as a *filter.* Coupling between the top-level process manager and the filter is designed to permit easy filter removal and replacement for experimentation with various protocols. Data moved vertically through the architecture is wrapped and unwrapped in layer envelopes similar to the way many network protocols function to provide independent services.

Figure 3 shows the major BATTLESIM, TCHSIM, and SPECTRUM components in the context of the general DES architecture (32:52).

3.2.2 BATTLESIM Application Layer and OO Design. The BATTLESIM application is an object-oriented discrete event battlefield simulation. Most aspects of BATTLESIM design do not incorporate any notion of parallelism since SPECTRUM handles all aspects of communication and synchronization between coarse-grained processes. BATTLESIM features designed to support graphics and interactive control are not used in this study.

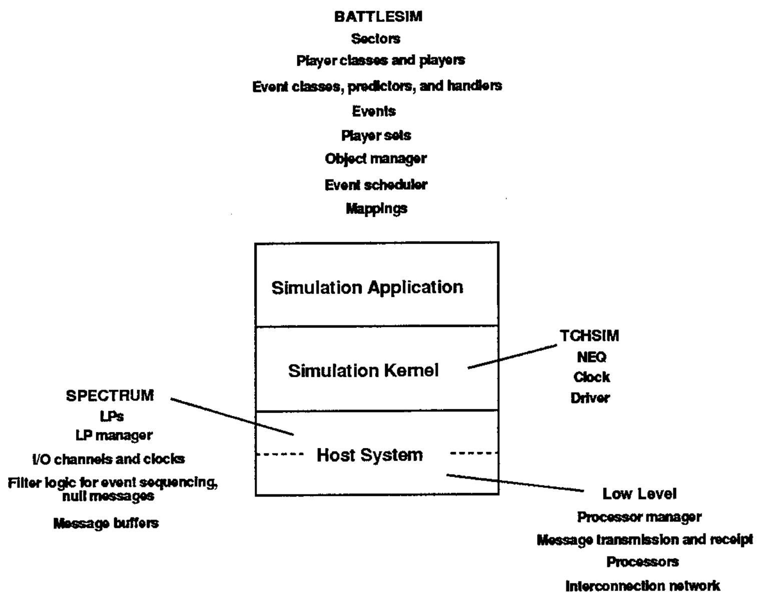

Figure 3. BATTLESIM, TCHSIM, and SPECTRUM

The mechanics of major design choices for BATTLESIM are explained in detail elsewhere (4, 51). However, an overview here provides the foundation necessary for the performance analysis.

3.2.2.1 Players and Player Classes. Battlefield entities are represented as player objects. Each player has a small common core of attributes, supplemented through inheritance by attributes used for battlefield modeling. These attributes capture the state of each player and include such features as player class, location, velocity, heading, and performance characteristics. A player's class describes a variety of characteristics shared with other players of the same class, thus establishing the types of behavior applicable to the player. The connection of behaviors to a player class is the basis for next event determination, or *prediction*, for a player in the class.

3.2.2.2 Events. Player actions, reactions, and interactions are represented as events occurring at particular times. The BATTLESIM design allows up to three players to be affected by one event, though two at a time is normally sufficient and is used in the performance models developed subsequently.

3.2.2.3 Event Classes. Event classes represent the behaviors modeled in BATTLESIM. Event classes capture actions, reactions, and interactions associated with each player class. Each event class is characterized by the types of players to which it applies, the number of players affected by an event instance, and the computation model of the behavior represented by the event class. Event instances in classes affecting a single player are *noninteractive*, while those in classes affecting two players are *interactive.*

3.2.2.4 Event Prediction. The interaction model in BATTLESIM uses the association of player classes with event classes along with current player state data to determine the next event for a player. When the next event for a player is needed, the prediction manager invokes an *event predictor* for each behavior that applies to the player. A predictor determines the time of the next instance of its associated event class for the player and returns the result as a temporary event to the prediction manager. If the event class is interactive, the temporary event contains a reference to the other player affected. The prediction manager filters the hypothesized result from each predictor and provides the event instance with the lowest time to the event scheduler. The event instance contains the time, event type, and identifier(s) of the player(s) involved.

The use of independent predictors maintains player state data encapsulation. Predictors are the only entities that have visibility to state data from more than one player at a time. This design facilitates integration of new player types. Similar approaches to event prediction for moving objects do not address the OO paradigm specifically, but do not appear to be significantly different in terms of efficiency (53).

3.2.2.5 Event Processing and Scheduling. The main simulation loop initiates calls to the prediction manager. Initially, the next event for each player is predicted and placed on the NEQ in time order. The loop then retrieves the next event from the queue and sends it to the handler associated with the event type. The handler executes the event, updating player state data as prescribed, and calls the prediction manager to determine the new next event(s) for the player(s) involved in the event just executed. After enqueuing the prediction result(s), the loop repeats until all events are processed or the maximum simulation time is reached. The prediction strategy limits the queue to holding at most one event per player at any time.

3.2.2.6 Player Management and Event Prediction with Sectoring. A cursory analysis of the event processing and scheduling processes shows that a significant amount of work is needed to predict a single interactive event. For an individual player, an interactive event predictor must use state data from all other players to determine the next event of that type for the player. Each of the interactive event predictors must do this, and the whole process is invoked for every event executed in the simulation.

Division of the battlefield into sectors is one way to reduce work associated with event prediction in both sequential and parallel simulation. However, the scheme introduces correctness and execution cost issues that don't exist in an unsectored design.

In a purely sequential simulation using sectoring, all player state data is immediately accessible on the processor. In contrast, if the simulation is to be run in parallel, some of the sector boundaries may coincide with LP boundaries. When a player of size passes over a boundary, a proper spatial mapping may place the player on two LPs, and thus two processors, at once. The player could then interact with other players on either processor while straddling the boundary.

Several choices are available to maintain player state data consistency and proper event prediction across LPs and processors. At one extreme, all player data can be replicated and updated in all LPs. On the other hand, an LP gaining a player can query the losing LP to send state data for just the player in question (17:21). A compromise solution was chosen for BATTLESIM (4:37). This approach is used when a player crosses any sector boundary, regardless of whether the boundary coincides with an LP boundary or not. If it doesn't, the process is applied in the same way except that events generated are scheduled on the local NEQ.

Front Crossing. When a player's front contacts a boundary, the losing LP sends a copy of the player to the gaining LP. The player is transmitted by generating an instance of a special, non-predicted event class, ADD_PLAYER_COPY. The event time is set for execution at the losing LP's current simulation time. Along with the time, the event contains a full copy of the player (4:36).

The gaining LP receives the event and places it in sorted order in its NEQ. The LP processes the event, creating a copy of the player from the data in the event and adding the copy to a special data structure. The LP treats the copy as any other player, using it in event predictions. Events predicted for the copy are sent to the losing LP, which remains designated as the owning LP until the center of the player crosses the boundary. The owning LP executes all events for the player, updating its state and sending state updates to any other LP that has a copy. Instances of another non-predicted event class, UPDATE_PLAYER_COPY, serve as the vehicle for passing state data to and causing state update on LPs with copies. The simulation time associated with state updates is the same as the time at which the original player is updated (4:36).

Center Crossing. When the player's center crosses into the gaining sector, the player image in the gaining LP becomes the original while the losing LP begins treating the image as a copy. Both LPs adhere to the player management scheme but with reversed roles, using UPDATE_PLAYER_COPY events to transmit state updates (4:36).

Back Crossing. When the player's back crosses into the new sector, the gaining LP generates an instance of a different non-predicted event class, REMOVE_PLAYER_COPY, and sends the event to the losing LP. The losing LP executes this event to remove its copy of the player image (4:36).

3.2.3 Design Limitations. The BATTLESIM design addresses many issues central to analysis of static properties of the problem. However, some limitations exist that place practical constraints on the scope of the analysis.

3.2.3.1 Loss of Detailed Functionality From OO Redesign. The primary emphasis of BATTLESIM is to model the behavior of objects moving and interacting in three-dimensional space. In this case, behavior modeling is intended to be representative of autonomous battlefield objects such as planes, tanks, and trucks that move according to physical laws involving space and time.

Early BATTLESIM versions (46, 37) contained a more fully functional implementation of the details of battlefield entities. The design and implementation was based on functional decomposition rather than the OO paradigm. The most recent work defined and implemented an object interaction model that could support parallel simulation using the OO paradigm (4, 51). Detailed functionalities such as range-limited sensors, evasive maneuvering precipitated by sensor contact, and combat engagement are not present in the OO design. These types of features affect the dynamic behavior of the simulation. Since the primary concern is the study of static problem attributes, little net loss accrues from their absence.

3.2.3.2 Sensor Capabilities. Sensors are typically simulated either as a solid or hollow projection of a player. With solid projection, interactions constantly occur between the player and objects within the projection range. With hollow projection, interactions occur when the projection boundary intersects another player. The original BATTLESIM design used a hollow projection.

The use of sectors affects sequential and parallel performance for both solid and hollow projection. In a sequential simulation using sectoring, sensor capability requires that all sectors covered by any part of the projection range be considered in player event prediction. This affects sequential performance by potentially increasing the number of players that must be used in event prediction. The frequency of event prediction for the player with the sensor also determines the cumulative effect on performance. In a parallel simulation, further impacts occur when sensor range boundaries cross LP space boundaries. The problem is analogous to the situation in which a player straddles processor boundaries for a period of time. Some method for shared state must be implemented. This forces the affected processors to run in complete synchronization for the duration of the crossing (53:917) (17:21).

The performance models in Chapter IV address the notion of shared state by distinctly modeling the number of players in each sector at individual simulation times. This provides a starting point performance impact analysis associated with sensor capabilities. However, the dynamic behavior generated by players having different sensor ranges that move with the player over time is not modeled.

3.2.3.3 Fixed Sector Adjacency. The spatial model upon which BATTLESIM is based restricts sector adjacency to at most two sectors along a single boundary. This structure precludes the use of sectors with irregular adjacencies, such as those that might be produced by a recursive bisection method popular in many parallel applications. Irregular adjacencies are normally used as part of load balancing schemes (3:571).

3.2.3.4 Simplified Event Scheduling Policy. The current design uses a simple scheduler that does not cancel superceded events. Event supercession occurs when event prediction for one player schedules an interactive event for that player, but the other involved player executes some other that nullifies the scheduled interactive event (53:917). Cancellation was implemented in earlier versions but not carried through the OO redesign. As with sensor capability, the effects of the scheduling policy fall primarily in the dynamic portion of the problem.

3.3 Definition Framework

This section establishes the primary data definition framework used to develop performance models for both the sequential and parallel battlefield simulation algorithms. Appendix A contains the complete set of definitions. The definitions are developed from the design study and reflect the principal data components of the application, simulation kernel, and host machine level.

To assist in capturing and exploiting group, or inheritance, relationships, several set partitions are established. Set partitioning based on equivalence classes is helpful in characterizing behavior across a class of related data items. Most of the partition definitions are abstract levels not explicitly used in the performance models and thus are found in the Appendix.

A particular simulation context establishes underlying relations formally captured in the data definitions. Dynamic relation membership is reflected in the performance models by discrete functions. Function values may be fixed, assumed, or left unknown. The use of discrete functions in the performance models allows for easy transition to nondeterministic models.

3.3.1 Notation and conventions. General set and algebraic symbols are consistent with usage defined by Stanat and McAllister (48). Naming conventions are established to reduce conflict among operators and semantics associated with set definitions.

- Sets are named with upper case Arabic letters. Names may be single letters, but normally use multiple letters for clarity and distinction from symbols used in the performance models.
- Set elements are named with one or more lower case Arabic letters found in the set name. Subscripted element names are used for reference via implicitly defined sequencing.
- For each countable set in the data model, there is an ordering bijection to $\mathcal{Z}^+$ that establishes denumerability. The details of the bijections are not significant for the performance models. For each such set A, the integers in the bijection are assumed to lie in $[1, |A|]$ unless otherwise noted.
- Relations and functions explicitly defined over sets are named with upper case Arabic letters.
- Functions representing abstract computation operations are named with lower case Greek letters to set them apart from functions explicitly defined over sets.
- Tuples forming relations are subscripted for distinguishability. Reference to a component of a tuple is made using the dot notation.
- Set cardinalities are represented with a single lower case symbol appearing in the name of the set. A cardinality symbol for a set is distinguished from the symbol representing an element of the set by the absence of a subscript: x for cardinality vice x_i to denote an element. Cardinalities of subsets of a particular set are distinguished from one another by judicious use of the $'$ mark.
- Arithmetic multiplication is shown explicitly with the $\times$ operator.

3.3.2 Definition Framework. The data definitions are outlined in terms of set composition rules, informal descriptions, structural constraints, and general restrictions and assumptions.

- $TIMES = \{x | x \text{ is a simulation time}\}$

  $TIMES$ is the set of simulation times projected or used in a simulation.

- $PLYR = \{\langle z, ATT_z \rangle | z \in \mathcal{Z}^+ \wedge ATT_z \subset \mathcal{P}(ATT) - \{\emptyset\}\}$

  $PLYR$ is the set of instances of simulated entities, hereafter called *players.* As a model of a physical component or system, each player has a *unique* identifier drawn from the positive integers and a single, non-null set of attributes. Attribute set commonalities may exist among players, providing the basis for player classes. Each player has an associated type that is the first attribute in the attribute tuple. Each player's attribute set remains static throughout its existence, though attribute values may change. The value of the type attribute for a given player remains constant during the player's existence.

- $BHVR = \{x | x \text{ is a simulated behavior}\}$

  $BHVR$ is the set of behaviors or processes that operate on physical components or systems modeled in the simulation. The number of behaviors modeled in a simulation must be finite. Members of $BHVR$ are modeled by iterative computational processes.

- $INTACT = \{TIMES \times (BHVR \cup \{\emptyset\}) \times (PLYR \cup \{\emptyset\})^2\}$

  $INTACT$ is the set of all possible applications of behaviors or processes that can be applied to up to two players at any time.

- $e \stackrel{\text{def}}{=} \langle t, b, p, q\rangle | \langle t, b, p, q\rangle \in TIMES \times (BHVR \cup \{\emptyset\}) \times (PLYR \cup \{\emptyset\})^2$

  e is the general form of a particular event in a DES.

- $EC = \{EC_x | x \in BHVR\}$

  EC is the set of all distinct event classes in a simulation.

- $PC = \{PC_x | x \in SIM\_ENT\}$

  PC is the set of all player classes distinguished by a type attribute value.

- $A' : EC \times PC \rightarrow \{0, 1\}$

  A' defines the association between event classes and player classes for a simulation:

$$A'(ec_i, pc_j) = \begin{cases} 1 & \text{if } ec_i \text{ applies to } pc_j \\ 0 & \text{otherwise} \end{cases}$$

- $A : EC \times PLYR \rightarrow \{0, 1\}$

  A defines the association between event classes and individual players in a simulation:

$$A(ec_i, p_j) = \begin{cases} 1 & \text{if } A'(ec_i, pc_k) = 1 \wedge p_j \in pc_k \\ 0 & \text{otherwise} \end{cases}$$

- $IN : PLYR \rightarrow SECTS$

  IN is a function mapping a player, either owner or copy, to the sector in which it currently resides. $|IN| = p$; $IN^{-1}(x)$ is the set of players in sector x; $|IN^{-1}(x)|$ is the number of players in sector x.

- $C : PLYR \rightarrow P' | P' \in \mathcal{P}(PLYR)$

C is a function mapping a player to copies of itself resident in other sectors. Elements of C are created when the front of a player crosses a sector boundary and are removed when the player's back crosses the same boundary.

- $m \overset{\text{def}}{=} \langle t, f, A\rangle | \langle t, f, A\rangle \in TIMES \times \{R\} \cup \{N\} \times \mathcal{P}(ATT)$

  m is the general form of a message in a PDES. t is the time of the message. f is the message type, either R or N for *Real* and *Null.* Real messages convey state data passed from one LP to another, while null messages distribute clock data among processors.

- $ch \overset{\text{def}}{=} \langle s, d, t, m\rangle | \langle s, d, t, m\rangle \in LP^2 \times TIMES \times MSGS \cup \{\emptyset\} \times \wedge m \neq \emptyset \Rightarrow t = m.t$

  ch is the general form of a communication channel between two LPs in a PDES where s is the source LP, d is the destination LP, and t is the current channel time. The channel time is set to be the time of the message currently in transit on the channel. If there is no message in transit, the channel time remains set to the time of the previous message.

- $IC_x = \{ch | ch \in CH_x \wedge x \in LP \wedge ch.d = x\}$

  IC_x is the set of all input channels defined for LP x.

- $OC_x = \{\langle ch, d\rangle | ch \in CH_x \wedge x \in LP \wedge ch.s = x \wedge d \in TIMES\}$

  OC_x is the set of all output channels defined for LP x. Each output channel has an associated delay time, d. The designator is distinct from the d destination attribute present at the base class level. The destination attribute is never used explicitly in subsequent modeling.

- $LP = \{\langle x, n, t, b, C\rangle | x$ is a logical process $\wedge\, n$ is a NEQ $\wedge\, t \in TIMES \wedge b$ is a buffer $\wedge$ $C = CH_x\}$

  This model uses a specialization of the general LP model (27:51) described on page 18. In addition to sequential code, message handling capability, and communication channels, each LP has a NEQ, a clock, and a message buffer. The NEQ and buffer are described on page 43. The sequential code corresponds to a particular simulation and (possibly null) interprocess synchronization algorithm. The message buffer contains all messages received by the LP that have not yet been processed.

- $ADJ : LP \rightarrow \mathcal{P}(LP)$

  ADJ is a function that maps an LP to its adjacent LPs. In this model, all LPs are adjacent to at least one other LP. Adjacent LPs have both an input and output channel going in each direction. This determines causal dependence.

3.4 Summary

The BATTLESIM application, TCHSIM simulation kernel, and SPECTRUM host machine service designs are analyzed in detail in this chapter. The analysis includes descriptions of significant data structures and application algorithms used to construct the simulation environment. A detailed description of the synchronization algorithm used in the parallel version of the simulation is presented in Chapter IV in the context of performance model development.

IV. Performance Model Derivation

This chapter introduces performance models for sequential and parallel battlefield DES involving interactive moving objects related in time and space. The performance models are developed over the framework of base set data definitions summarized in the previous chapter and defined in detail in Appendix A. Abstract data types, primitive abstract functions, and canonical algorithm models are used to formulate the sequential and parallel performance models.

The goal of performance modeling is to capture relationships among significant parts of the simulation design. Thus, the models are defined deterministically in terms of an arbitrary simulation with observable or measurable parameters that bind the sizes of the sets defined. These sizes are expressed as unspecified constants or computable functions that are unique to a particular simulation scenario. Algorithm models used as the basis for the performance models are representative of the battlefield simulation design described in Chapter III.

For each algorithm, the operations to be accomplished are modeled as primitives in terms of elements from the sets in the data definitions. Computation processes are modeled as abstract functions defined in this section. Discrete functions or relations capture the relationships established in the data definitions. Resulting primitives are combined through the use of summations to capture iterative behavior over data structures. Set cardinalities provide limits of summation. Summations are then combined through addition or multiplication as appropriate for the structure of the algorithm.

4.1 Performance Model Definitions and Descriptions

The performance models express the behavior of the canonical algorithms quantitatively by reference to data elements, relation membership, and algorithm structure. Set and relation membership and cardinality are the primary vehicles for translation between the abstract collective view found in the data definitions and the iterative, elemental view needed in the performance model. Additional definitions used in the performance models are cardinality expressions and abstract computation functions describing the work outlined in the algorithm.

4.1.1 Cardinality and Arithmetic Definitions.

All cardinalities are parameters taken from the simulation of interest. With the exception of the number of events in the simulation, s', all are observable or measurable at some point prior to completion of the simulation. As a practical matter, the value of s' is not known prior to termination of simulation represented by SIM.

- $c' = |IND|$, the number of noninteractive event classes.
- $c'' = |INTER|$, the number of interactive event classes.
- $c = c' + c'' = |EC|$, the total number of event classes.
- $p = |PLYR|$, the number of players.
- $s' = |SIM|$, the number of events in the simulation.

4.1.2 Reference Conventions.

Individual set elements are referenced with subscripts resolved by summation indices.

- p_i refers to the ith player $p \in PLYR$.
- ec_i refers to the ith event class $ec \in EC$.
- e_i refers to the ith event $e \in SIM$.
- $\tau(op)$ refers to the real time needed to perform operation *op*.
- $\tau_{s,i}$ refers to the real time needed to perform the ith iteration of step s where the step is referenced to an algorithm model. Steps not falling in loops are referenced as τ_s.
- LP_x refers to the logical process labeled x.
- EC ordering. Noninteractive event classes are ordered so as to precede interactive event classes. Special class ordering is unspecified.

4.1.3 Abstract Data Types and Function Definitions.

The primary abstract data types used in the models are the NEQ and message buffers. Every LP has one of each. A NEQ is a generic priority queue with defined but unspecified operations. Each operation completes in finite, measurable time and thus can be provided as an argument to an appropriate measurement function. A buffer is essentially identical to a NEQ except that it holds queued interprocessor messages rather than events. Each LP in a parallel simulation has a message buffer that holds messages coming from the input channels associated with the LP. Messages are maintained in a buffer in nondecreasing order by time.

Several functions are defined for both data types. Functions subscripted with n operate on a NEQ and events, while functions subscripted with b operate on a buffer and messages.

- ι_n, ι_b - Insert item into structure. This representation does not explicitly model structure operation efficiency. Dependency on simulation progress is introduced when needed by use of additional subscripts.
- γ_n, γ_b - Remove and return the first item from structure.
- ζ_n, ζ_b - Delete item from structure.

4.1.4 Operation Definitions. An untyped function, $\tau(x)$, represents the real simulator time or work to do x. Typed functions model particular primitive operations in the canonical algorithms. In each definition, $p_x \in PLYR$ and $e_i \in SIM$.

- $\chi(ec_i, p_j)$ and $\chi(ec_i, p_j, p_k)$ - Calculate next instance of event class ec_i for player p_j or for player p_j with respect to player p_k for $p_j, p_k \in PLYR$.
- $\delta(p_i)$ - Determine next event for player p_i.
- $\eta(ec_j)$ - Execute an event e_i in class ec_j for players $e_i.p$ or $e_i.p$ and $e_i.q$ as specified. This is an event class-wide worst case execution.
- $v(P\_SET), P\_SET \subset PLYR$ - Update the players in P_SET. Used to update player copies to maintain consistency with respect to simulation time.
- $\mu(C\_SET, i, g), C\_SET \subset CH$ - Return the minimum time of the channels in C_SET with respect to loop i and protocol iteration g.
- $\sigma(N), \sigma(R)$ - Send a real or null message between two processors. This function represents the activity needed for the sending process to send the message and be able to resume further processing. Message transit time on the interconnection network is not modeled.

- $\alpha(p_i) = IN^{-1}(IN(p_i))$ - Shorthand notation to construct the set of players, including copies, in the same sector with player p_i.
- $\omega(ADJ(x), i, g)$ - Denotes LP x waiting on receipt of null messages from its adjacent LPs as part of the gth input protocol iteration before processing the ith event.

4.2 Sequential Simulation

4.2.1 Canonical Sequential Simulation Algorithm.

The basic sequential simulation algorithm, presented in Figure 4, is a simple loop that retrieves the next event; executes it by updating the state of, and determining the next event(s) for, the player(s) involved; and schedules the next event(s) on the queue in time order. The loop continues until the queue is empty or an **End** event is executed.

The algorithm captures both the unsectored and sectored approaches. The unsectored approach is a specialized subset in which there is one sector and the number of players per sector is the number of players in the simulation. As a result, there are no copies generated. Procedural differences are shown parenthetically.

The algorithm description includes expansions of the processes used to determine and schedule an event for a player. Scheduling in step 1 assumes that no event cancellation occurs. Descriptive expansions occur once and then are included by reference. The algorithm assumes the existence of a deterministic resolution scheme to order events occurring at the same simulation time.

4.2.2 Sequential Performance Model.

Performance model development captures several aspects of each simulation iteration. These include event prediction, player copy

```
begin simulation
  Step 1.  For each player:
    Determine next event;                                  1.1
      Expansion:
        For each noninteractive event class:              1.1.1
          if the event class applies to the player:       1.1.2
            calculate next event class instance;          1.1.3
        For each interactive event class:                 1.1.4
          if the event class applies to the player:       1.1.5
            For each other player (in the sector):        1.1.6
              if the event class applies to the other player:  1.1.7
                calculate next event class instance       1.1.8
    Insert determined event in queue in time order         1.2
Loop:
  Step 2.  Remove and return first event; sim time = event time;
  Step 3.  Execute event;
  Step 4.  For each player affected by event:
    (Update copies;                                        4.1
      Expansion:
        For each copy:                                    4.1.1
          Insert update event in NEQ;                     4.1.2)
    Determine next event;                                  4.2
    Schedule next event;                                   4.3
      Expansion:
        If no event exists on the queue for this player (and   4.3.1
          its copies) with earlier time than determined event:
          Insert determined event;                        4.3.2
          If there is an event on the queue for this player    4.3.3
            (or its copies) with later time than
            determined event:
            Delete subsequent event;                      4.3.4
            If subsequent event affected another player (or    4.3.5
              its copies):
              Step 4 for other player;                    4.3.6
until END or queue empty
end Loop;
end simulation.
```

Figure 4. Canonical Sequential Algorithm

management, and simple NEQ operations. The performance model does not reflect complex NEQ operations such as event cancellation and rescheduling, or continuing overheads that accompany these types of operations. The performance effects of these types of operations depend largely on the implementation chosen for the NEQ and the dynamic behavior of the simulation (11) (9).

4.2.2.1 Actions Prior to Loop. $\tau(\chi(ec_i, p_j))$ denotes the time to calculate the next instance of event class i for player j, corresponding to step 1.1.3. $A(ec_i, p_j)$ specifies the applicability of event class i to player j, corresponding to step 1.1.2. Since there are c' noninteractive event classes, calculation of the next instance of each of them for a target player denoted p_j is modeled as

$$\sum_{i=1}^{c'} A(ec_i, p_j) \times \tau(\chi(ec_i, p_j)) \tag{1}$$

corresponding to the total of 1.1.1–1.1.3.

For each interactive event class, calculation of the next instance involves all pairings of the target player p_j and all other players in the same sector that can be affected by the event class. $\tau(\chi(ec_i, p_j, p_k))$ represents the time to calculate the next instance of event class i for player j with respect to player k, corresponding to step 1.1.8. $A(ec_i, p_k)$ is the applicability of event class i to player k, corresponding to step 1.1.7. $\alpha(p_j)$ is the set of players in the same sector as player j. Computation of event instances in the class for each of the other players, corresponding to steps 1.1.6–1.1.8, is represented by

$$\sum_{p_k \in \alpha(p_j)} A(ec_i, p_k) \times \tau(\chi(ec_i, p_j, p_k)) \tag{2}$$

In implementation, $p_k = p_j$ would not be chosen, reducing one iteration from this expression.

$A(ec_i, p_j)$ specifies the applicability of event class i to player j, corresponding to step 1.1.5. Iteration over all $c - c'$ interactive event classes, corresponding to steps 1.1.4–1.1.8, is represented by

$$\sum_{i=c'+1}^{c} A(ec_i, p_j) \times \left[\sum_{p_k \in \alpha(p_j)} A(ec_i, p_k) \times \tau(\chi(ec_i, p_j, p_k)) \right] \tag{3}$$

The time to determine the next event for a particular player p_j, denoted by $\tau(\delta(p_j))$ and corresponding to step 1.1, is expressed by combining (1) and (3):

$$\begin{aligned} \tau(\delta(p_j)) &= \sum_{i=1}^{c'} A(ec_i, p_j) \times \tau(\chi(ec_i, p_j)) + \\ &\quad \sum_{i=c'+1}^{c} A(ec_i, p_j) \times \left[\sum_{p_k \in \alpha(p_j)} A(ec_i, p_k) \times \tau(\chi(ec_i, p_j, p_k)) \right] \end{aligned} \tag{4}$$

When the next event for the jth player is determined, events for $j - 1$ players have been placed in the queue. $\tau(\iota_{n,j-1})$ represents the time needed to insert the event for the jth player, corresponding to step 1.2. Combining expressions for steps 1.1 and 1.2 results in the time τ_1 needed to determine and schedule the next event for all p players, corresponding to step 1:

$$\tau_1 = \sum_{j=1}^{p} [\tau(\delta(p_j)) + \tau(\iota_{n,j-1})] \tag{5}$$

4.2.2.2 Model for Loop.

Dynamic NEQ Length and Searches. Upon entering the loop to process the first event in the simulation, the NEQ holds p events since one event was scheduled for each of p players. Scheduling and processing noninteractive events results in the replacement of each event executed, causing queue length to vary trivially from p to $p - 1$.

Player interaction results in dynamic NEQ length. When a scheduled event for one player is executed, a new event is predicted and scheduled. If the new scheduled event is interactive, the existing but as yet unprocessed event for the other player must be cancelled. Interaction is a function of simulation progress and, in general, can result in significant variance in NEQ length, from the initial length p to any integral value down to $\lceil p/2 \rceil$. While shorter queue lengths may result in less time spent on queue operations, the overhead added by queue searches and event comparisons may offset this. Other queue searches are needed on every scheduling operation to preserve consistency when multiple events are scheduled for the same simulation time.

The algorithm model reflects cancellation and replacement. For simplicity, the performance model does not include these operations explicitly. However, NEQ insertion during each loop iteration is modeled as a function of the previous loop iteration. Representation of this dependency maintains an accurate abstraction regardless of the efficiency or implementation characteristics of the NEQ.

Static Performance Model. The time needed to remove and return the first event from the queue, denoted $\tau_{2,i}$ and corresponding to step 2, is:

$$\tau_{2,i} = \tau(\gamma_n) \tag{6}$$

Execution time for an event instance e in some event class ec_j is dependent on the computation model of behavior simulated by the event's class and the number of players affected by events in that class. The time is represented as $\tau(\eta(ec_j))$. An event instance can belong to only one class. Summation over the event classes using a conditional test provides an explicit representation of mutual exclusion among event classes. The time $\tau_{3,i}$ to execute a particular event e_i in class ec_j corresponding to step 3, is

$$\tau_{3,i} = \sum_{j=1}^{c}(e_i \in ec_j) \times \tau(\eta(ec_j)) \tag{7}$$

Player copies in other sectors must be updated for each player affected by the event. The update action corresponds to step 4.1. The set of copies associated with player p is $C(p)$. For each copy, an update event is inserted in the NEQ with time set to the current simulation time. Thus, the extra work needed for update is a function of NEQ insertion time, leading to an expression for $\tau(v(C(p)))$, corresponding to steps 4.1.1–4.1.2:

$$\tau(v(C(p))) \geq |C(p)| \times \tau(\iota_{n,i-1}) \tag{8}$$

This expression is left as a lower bound since the NEQ insertion time is not modeled to the granularity of multiple event insertion on a single simulation loop iteration.

The time to update one or two players, depending on how many are involved in an event e_i, is represented by $\tau(\upsilon(C(e_i.p))) + ((e_i.q \neq \emptyset) \times \tau(\upsilon(C(e_i.q))))$, corresponding to step 4.1.

Next event determination occurs either once or twice, depending on whether event e_i is interactive. The time needed to determine the next event for the players affected by event e_i, corresponding to step 4.2, is

$$\tau(\delta(e_i.p)) + ((e_i.q \neq \emptyset) \times \tau(\delta(e_i.q))) \tag{9}$$

where $e_i.p$ and $e_i.q$ refer to the first and second players affected by the event using the dot notation to reference tuple members. Step 4.3 is represented by $\tau(\iota_{n,i-1})$, the time to insert an event in the queue. As with copy updates and event determination, insertion will happen either once or twice, depending on the number of players involved in the event being executed. The time to insert is $\tau(\iota_{n,i-1}) + ((e_i.q \neq \emptyset) \times \tau(\iota_{n,i}))$.

Combining expressions for steps 4.1, 4.2, and 4.3 gives $\tau_{4,i}$, an expression for the time to update copies, determine the next event, and schedule the next event for each player possibly affected by event e_i, corresponding to step 4:

$$\begin{aligned}\tau_{4,i} &= \tau(\upsilon(C(e_i.p))) + \tau(\delta(e_i.p)) + \tau(\iota_{n,i-1}) + \\ &\quad (e_i.q \neq \emptyset) \times (\tau(\upsilon(C(e_i.q))) + \tau(\delta(e_i.q)) + \tau(\iota_{n,i}))\end{aligned} \tag{10}$$

Combining $\tau_{2,i}$, $\tau_{3,i}$, and $\tau_{4,i}$ results in the time to completely process event e_i corresponding to a loop iteration:

$$\begin{aligned}\tau_{2,i} + \tau_{3,i} + \tau_{4,i} \quad = \quad & \tau(\gamma_n) + \left[\sum_{j=1}^{c}(e_i \in ec_j) \times \tau(\eta(ec_j))\right] + \\ & \tau(v(C(e_i.p))) + \tau(\delta(e_i.p)) + \tau(\iota_{n,i-1}) + \\ & (e_i.q \neq \emptyset) \times (\tau(v(C(e_i.q))) + \tau(\delta(e_i.q)) + \tau(\iota_{n,i})) \qquad (11)\end{aligned}$$

The loop iterates s' times, corresponding to the number of events in the simulation. Combining τ_1 with the total time spent in the loop gives an expression for sequential finishing time, τ_{seq}:

$$\tau_{seq} = \tau_1 + \sum_{i=1}^{s'}(\tau_{2,i} + \tau_{3,i} + \tau_{4,i}) \qquad (12)$$

4.2.2.3 Complexity Analysis. The complexity of most simulations is related to the number of players, p. In a simulation with one or more interactive event predictors, the generation of each of the s' events requires p^2 player comparisons. If there is at least one event per player in the simulation, $s' \geq p$. Thus, the complexity of such a simulation is at least $O(p^3)$. This is a best case complexity as well, since all comparisons must be made. Thus, the complexity can be expressed as $\theta(p^3)$.

Different simulations may have different numbers of events. For simulations in which $s' > p^2$, the complexity is $\theta(p^4)$. This progression holds as s' grows.

To decrease sequential simulation execution time for a particular simulation, the number of players involved in each interactive event class prediction cycle must be reduced. No other reduction opportunity exists, since neither the number of players or events in the simulation can be decreased without altering the problem being simulated.

4.2.2.4 Estimating the Number of Sector Crossings. The cost of sectoring arises from execution of boundary crossing events, additional prediction cycles caused by crossings, and prediction over player copies duplicated while players straddle boundaries. The number of sector crossings can be estimated by examining all player routes in the simulation. Each moving player has a route as an attribute. A route is a finite sequence of points in 2D space of length l.

For a battlefield of width W distributed uniformly over r' sectors in one dimension, each sector is of width $w = W/r'$. The number of times a route crosses boundaries perpendicular to a given axis and occurring every w units can be determined.

Let $bx_{j,w}$ denote the number of boundary crossings for the route of player p_j with boundaries placed every w units along the x axis. Each point in the route is a tuple of the form $\langle x, y \rangle$. Then

$$bx_{j,w} = \sum_{i=1}^{l-1} \left| \lfloor \frac{x_i}{w} \rfloor - \lfloor \frac{x_{i+1}}{w} \rfloor \right| \tag{13}$$

For all players in the simulation, the number of boundary crossings, denoted bx_w, is

$$bx_w = \sum_{j=1}^{p} bx_{j,w} \tag{14}$$

Crossings of boundaries that cut the y axis are tabulated in a similar fashion, substituting y for x and using the y values of route points.

The additional cost of prediction over players and player copies in each sector is already embedded in the model by treating player copies as regular players and updating player copies on every iteration. However, there must be at least one additional event for each boundary crossing to trigger data structure reorganization. The actual number of events per crossing is an implementation detail. The approximate cost of sectoring is reflected by adjusting the limit of summation for loop execution by the number of additional events due to sectoring. Using $\mathcal{X}$ to denote the number of events per crossing gives a total number of events $\mathcal{X}bx_w$. Combining this result with τ_1 gives an expression for sequential finishing time, τ_{seq}:

$$\tau_{seq} = \tau_1 + \sum_{i=1}^{s'+\mathcal{X}bx_w} (\tau_{2,i} + \tau_{3,i} + \tau_{4,i}) \tag{15}$$

4.3 Parallel Simulation

4.3.1 Canonical Parallel Simulation Algorithm. In the SPMD parallel simulation approach, identical copies of the sequential algorithm are placed on each available processor. Data is initially supplied to each processor in accordance with the chosen decomposition scheme. In this case, battlefield sectors and the players they contain are placed on each processor. The sector-to-processor mapping remains static throughout the simulation. Players migrate among processors as they move between sectors mapped to different processors.

4.3.1.1 Algorithm Abbreviations. Several abbreviations in the algorithm description clarify its presentation.

- **msg** - message, either real or null.
- **(i)(o)chan** - input or output channel respectively.
- **delay** - propogation delay associated with LP.
- **min** - arithmetic minimum.

```
begin simulation
  Step 1.  For each player:
    Determine next event;                                           1.1
      Expansion:
        For each noninteractive event class:                       1.1.1
          if the event class applies to the player:                1.1.2
            calculate next event class instance;                   1.1.3
        For each interactive event class:                          1.1.4
          if the event class applies to the player:                1.1.5
            For each other player in the prediction range:         1.1.6
              if the event class applies to the other player:      1.1.7
                calculate next event class instance                1.1.8
    Insert event instance with minimum time (determined event)      1.2
      in queue in order
```

Figure 5. Canonical Parallel Algorithm - Initial Step

Figure 5 reiterates the initial portion of the algorithm prior to the main simulation loop. Figure 6 introduces the input and output portions of the synchronization protocol into the main loop.

As an abstract algorithm, the description does not include a specific input message buffer or buffer management. The algorithm assumes the presence of an infinite input buffer into which messages arriving from input channels are placed in time order. Buffering techniques are normally used in implementation to mollify the effects of communication latency (27:60).

4.3.1.2 Protocol Description. Each sequential process on a processor is a *Logical Process (LP)*. Each LP has a *Next Event Queue (NEQ)*, a local clock, input channels from predecessor, or *upstream*, LPs, and output channels to successor, or *downstream*, LPs. The output channels from an LP are the input channels to its successor LPs. Each channel has an associated timestamp set to the time of the last message that traversed the channel.

```
Loop:
  Step 1A.  Input protocol
    Expansion:
      Loop Inner:
        Case (real msg):
          Update msg chan time to msg time;
          Insert msg data in NEQ;                                      1A.1
        Case (null msg):
          Update msg chan time to msg time;
          Discard null msg;                                            1A.2
        If (NEQ time <= min (ichan times)):
          exit Loop Inner;                                             1A.3
        else
          Send nulls on ochans with msg time =                         1A.4
            min (NEQ time, (min (ichan times) + ochan delay)));
          Wait for msg;                                                1A.5
      end Loop Inner;
  Step 2.  Remove and return first event; sim time = event time;
  Step 3.  Execute event;
  Step 4.  For each player affected by event:
    Update copies;                                                      4.1
      Expansion:
        For copies on this LP:                                        4.1.1
          Insert update event in NEQ;                                 4.1.2
        For copies on other LPs:                                      4.1.3
          Send update events to other LPs;                            4.1.4
    Determine next event;  (1.1.1 - 1.1.8)                              4.2
    Schedule next event;                                                4.3
      Expansion:
        If player is on another LP:                                   4.3.1
          Send event to other LP with current time;                   4.3.2
          Send null messages on all remaining output chans;           4.3.3
        else
          Schedule according to cancellation policy                     4.4
until (END executed or queue empty)
end Loop;
end simulation.
```

Figure 6. Canonical Parallel Algorithm

Channels obey a First-In First-Out discipline, and messages arriving on a channel have monotonically increasing timestamps.

The objective of the protocol is to ensure that every event removed from the NEQ and processed by the LP is in fact the next event that could possibly be executed. If an event-bearing message could arrive with an earlier timestamp, the LP must wait until that message actually arrives.

Input Protocol. The input portion of the protocol is exercised prior to every attempt to retrieve and execute the next event from the NEQ. The behavior of the protocol, both on entry and on each protocol iteration once entered, is dependent on whether or not any messages have been received since the last protocol iteration. Within this dependency, protocol behavior is based on the relationship between the times of messages received over input channels and the time of the first event on the NEQ. The order of evaluation in the protocol reflects the inherent dependency on input channel state change.

If there is a message in the input buffer, the input channel times are updated to reflect the time of the message. If the message is a real message, it is inserted in the NEQ for eventual execution.

After the buffer check, the next step is to decide whether any NEQ events are safe to process as indicated by the first event time being less than the minimum input channel time. If any are safe, the first event must be safe. Thus, the protocol exits and the event is returned from the NEQ. Otherwise, none are safe, so the LP reports the next possible time it could send an event to its downstream LPs. This time is either the time of the first event on the NEQ, since that event could be a boundary crossing, or the minimum input channel time plus the LP delay, since the incoming message at the lowest input channel time could be a boundary crossing.

If there is no message in the buffer, the LP must wait for one to arrive. The process is then repeated with the new information provided by the message.

Output Protocol. The output portion of the protocol is less complex than the receiving portion. When the LP sends a real message to a downstream LP, it sends a null message with the same timestamp to each of its other downstream LPs as long as sending the null increases the channel time. The real and null messages both contain the LP's current simulation time which was set when executing the event. This time transmission increases the input channel time of all downstream LPs, possibly allowing them to execute some number of events.

The output portion of the protocol does not significantly affect the model already established for copy updating. The iterative behavior of copy updating is addressed as a portion of the sequential part of the algorithm. Since message transmission is asynchronous, the sending LP does not wait for message receipt. All waiting time is modeled in the input protocol model for a receiving LP. Since the receiving LP does not know whether the next communication it is waiting for is a real message or a null message, no distinction is made for waiting periods based on message type.

4.3.2 Input Protocol Model. The input protocol model evolves somewhat differently than the sequential model since the parallel algorithm involves a dynamically determined, mutually exclusive condition over two factors, one of which is dependent on the simulation scenario (steps 1A.1 and 1A.2); two steps that can iterate many times depending on the simulation time progress of other LPs (steps 1A.4 and 1A.5); and one step with a real-time component τ that is a dynamic composite function of the total real-time components of other LPs (step 1A.5).

Component expressions are developed for a single occurrence of each factor and grouped in a conditional model. Logical properties determining protocol iteration bounds within each event processing cycle are used to establish potential upper bounds on iteration.

4.3.2.1 Input Protocol Initial State. When the input protocol for LP x commences, the state of LP x and its adjacent LPs can be described as follows:

- Let $E_{x,i} = \langle e_{x,i}, e_{x,i+1}, \ldots, e_{x,i+n} \rangle$ denote the sequence of events in the NEQ for LP x prior to processing event $e_{x,i}$ where i corresponds to the ith iteration of the main loop for LP x.
- Let $E_{z,j} = \langle e_{z,j}, e_{z,j+1}, \ldots, e_{z,j+o} \rangle$ denote the sequence of events in the NEQ for an arbitrary LP $z \in ADJ(x)$ when LP x enters the protocol prior to processing its next event. In this case, the index j corresponds to the jth iteration of the main loop for LP z.
- $LP_x.t_i = e_{x,i-1}.t$ and $LP_z.t_j = e_{z,j-1}.t$. The simulation time for an LP is the time of the last event executed by the LP.

Index g is used when necessary to distinguish among input protocol iterations for a particular LP. Values for g always begin with 1 prior to execution of the ith event on LP x. The index is incremented for every return to **Loop Inner** prior to exiting the loop. Where necessary, expressions for minimum input channel times and waiting times are expressed in terms of g to distinguish between different protocol iterations. If distinction is needed among input protocol progress for two LPs at the same time, g' is used as well. Usage is clear in context.

4.3.2.2 Individual Components. Each input protocol iteration has two behaviorally independent phases. In the first phase, the message buffer is checked. In the second phase, the time of the first event on the NEQ is compared to the current minimum input channel time, $\mu(IC_x, i, g)$. Let $LP_x.\tau_{1A,i,g,1}$ and $LP_x.\tau_{1A,i,g,2}$ denote the time according to each phase. Several cases are possible in each phase.

- Phase 1.

  1. *Qualified Real Message Present.* $LP_x.b$ represents the first message buffer position for LP x. If there is a message m in that position and $m.f = R$, the message is carrying an event to be inserted in the NEQ. The time for the input channel on which the message arrived is updated to the message time, the message is removed from the buffer, and the message data is inserted in the NEQ. The time to remove the message and insert the message data is represented as $\tau(\gamma_b) + \tau(\iota_{n,i',g-1})$ and corresponds to step 1A.1. If $g = 1$, $i' = i - 1$ and g is referenced to the last protocol iteration of the previous loop iteration. Otherwise, $i' = i$. This reflects the dependence of the time to insert an event on the state of the NEQ at the completion of the previous input protocol iteration.

  2. *Null Message Present.* If $LP_x.b = m$ and $m.f = N$, indicating that a null message is present in the buffer, the time for the input channel on which the

message arrived is updated to the message time. The message itself is removed and destroyed. This action is represented as $\tau(\gamma_b) + \tau(\zeta_b)$ and corresponds to step 1A.2.

- Phase 2.

  1. *Qualified NEQ Event Present.* The first event in the NEQ is represented by $e_{x,i}$. If $e_{x,i}.t \leq \mu(IC_x, i, g)$, the event can be processed. No significant time is spent in the loop. This corresponds to step 1A.3. The time for event retrieval is modeled for step 2 outside the protocol.

  2. *Qualified NEQ Event Not Present.* If $e_{x,i}.t > \mu(IC_x, i, g)$, the first event on the NEQ cannot be processed. If null messages with the current null message time have not already been sent to neighboring LPs, they must be sent to maintain simulation progress. The time needed to transmit and subsequently wait for new null messages is modeled as part of step 1A.5.

  3. *Awaiting Message.* Once null messages are sent, the LP must wait for some new message to arrive in its buffer. The minimum amount of time that LP x must wait is $2 \times \tau(\sigma(N))$, the time needed for two neighboring LPs to send null messages to one another in direct succession. If no neighbor is waiting to receive the null and return one, then LP x must wait $2 \times \tau(\sigma(N)) + \tau(\omega(ADJ(x), i, g))$, corresponding to step 1A.5.

The resulting components are summarized for the gth input protocol iteration during the ith event iteration for LP x:

$$LP_x.\tau_{1A,i,g,1} = \begin{cases} \tau(\gamma_n) + \tau(\iota_{n,i',g-1}) & \text{if } LP_x.b = m; m.f = R \\ \tau(\gamma_b) + \tau(\zeta_b) & \text{if } LP_x.b = m; m.f = N \end{cases} \tag{16}$$

and

$$LP_x.\tau_{1A,i,g,2} = \begin{cases} 0 & \text{if } e_{x,i}.t \le \mu(IC_x, i, g) \\ 2 \times \tau(\sigma(N)) + \tau(\omega(ADJ(x), i, g)) & \text{if } e_{x,i}.t > \mu(IC_x, i, g) \end{cases} \tag{17}$$

The time for the iteration is expressed as the sum of its two phases:

$$LP_x.\tau_{1A,i,g} = LP_x.\tau_{1A,i,g,1} + LP_x.\tau_{1A,i,g,2} \tag{18}$$

4.3.2.3 Bounds on Null Message Iteration. The number of null receipts, input channel time updates, and null transmissions during the ith event iteration is denoted $h_{x,i}$ and determined by several factors depending on the operation of adjacent LPs.

The timestamp of a null message sent by an LP $z \in ADJ(x)$ before processing event $e_{z,j}$ is derived from one of three values. If $\mu(IC_z, j, g') + OC_z.d_x < e_{z,j}.t$, $\mu(IC_z, j, g') + OC_z.d_x$ is used. If the reverse is true, $e_{z,j}.t$ is used. The former represents the time at which the earliest possible incoming message, real or null, could generate some outgoing message. The latter represents the time of the first event on the NEQ. The lowest of these two times is the earliest time the LP could generate an outgoing message.

The third possible value for time applied to the outgoing null message is the current simulation time. If the null message is generated as a result of LP z sending a real message to some other LP y, the time of the real message is used for the null message as well.

The three choices present several possibilities for $h_{x,i}$. Consideration of behavior over multiple protocol iterations establishes assumptions for analysis.

If two adjacent LPs each go through a protocol iteration and report their next event times to one another, each will be able to execute an event after the other on subsequent iterations. If this continues repeatedly, execution consists of cycles of null message exchange followed by sequenced pairs of event executions. The other extreme is the continual exchange of messages using only minimum input channel times incremented by fixed delays as timestamps on null messages. For an event to be executed, this must eventually be ended by one LP reporting its next event time to the other. Thus, the analysis focuses on the situation in which the channel delay time increment is sufficiently small with respect to initial minimum input channel times and next event times on both LPs so as to cause some number of null messages to be generated.

Let $\Lambda_{x,i}$ denote the simulation time interval between the current simulation time and the next event time for LP x, $e_{x,i-1}.t$ and $e_{x,i}.t$ respectively. Since the ith iteration for LP x begins after execution of event $e_{x,i-1}$, the simulation time for LP x is the time of the event just executed: $LP_x.t = e_{x,i-1}.t$. Likewise, let $\Lambda_{z,j}$ denote the simulation time interval from $e_{z,j-1}.t$ to $e_{z,j}.t$.

After processing event $e_{x,i-1}$, LP x must enter its input protocol loop before processing event $e_{x,i}$. By construction, $\mu(IC_x, i, 1) < e_{x,i}.t$. Thus, at least one new message must arrive before LP x can process its next event. Let z denote an LP in $ADJ(x)$ such that $OC_x.d_z$ is the minimum for all output delays on channels from LP x. Within $|ADJ(x)|$ input protocol iterations, LP z must send a message to LP x which may or may not allow event $e_{x,i}$ to be processed.

Again by construction, LP x is unable to process its next event after receiving the message from LP z, so $\mu(IC_z, j, g') + OC_z.d_x < e_{x,i}.t$. LP x always adds $OC_x.d_z$ to the time of any null received from LP z. If LP z sends no real messages to some other LP y adjacent to it, any null received by LP x represents some $\mu(IC_z) \leq e_{z,j}.t$. Further, if $OC_z.d_x = 0$, then $h_{x,i} = \frac{\Lambda_{x,i}}{OC_x.d_z}$. An expression for $h_{x,i}$ for LP x with LP $z \in ADJ(x)$ that sends no real messages to other LPs during the interval $\Lambda_{x,i}$ is:

$$h_{x,i} \geq |ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \qquad (19)$$

The inequality accounts for LPs in $ADJ(x)$ other than z possibly sending a null message to x as a result of sending a real message to some other LP. If this happens, the null message sent increments $\mu(IC_x)$ by an amount less than $OC_x.d_z$. This slows the rate at which $\mu(IC_x)$ advances toward $e_{x,i}.t$. As the difference between the normal increment and smaller increments accumulates over the interval $\Lambda_{x,i}$, the total number of nulls sent and received by LP x may increase over the number that would be sent and received if this behavior did not happen. The aperiodic jumps in $\mu(IC_x, i, g)$ cause an effect similar to *jitter delay* (43:75).

Extra null messages received by LP x for each real message LP z sends to some LP other than x during the interval denoted by $\Lambda_{x,i}$ can be captured by using Equation 19 as the lower bound of a range for $h_{x,i}$ and establishing the upper bound. Let $m_{z,x,i}{}'$ denote the number of real messages sent by LP z to LPs in $ADJ(z)$ other than x during this interval. The range for $h_{x,i}$ is expressed as:

$$|ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \leq h_{x,i} \leq |ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} + m_{z,x,i}{}' \qquad (20)$$

The upper bound term can be further expressed as a summation over the LPs in $ADJ(x)$ to model the effect of those LPs cumulatively. Since this factor is inherently dynamic based on the simulation scenario, it is left as a simple term in further treatment to preserve model clarity. The actual value of $m_{z,x,i}{}'$ is a multiple of the number of boundary crossings occurring either into or out of LP z during the interval $\Lambda_{x,i}$. For each boundary crossing, LP z sends some application-specific number of messages, but at least one, to the gaining LP y to establish or terminate shared state data maintenance. Since the number of boundary crossings in any interval is problem dependent and based on cumulative player movement, it is not modeled in further detail. The estimation approach for sector crossings described on page 53 provides one method for determining the actual number of crossings for a deterministic trace-driven simulation.

The range expression for $h_{x,i}$ shows algebraically why a cycle among LPs must have an additive nonzero delay in order to prevent deadlock. As the additive delay approaches zero, the number of inter-event increments for each of the LPs in the cycle grows without bound. Thus, the next event for each LP becomes unreachable.

4.3.2.4 Waiting Time. The time LP x spends waiting to receive a null message is also developed without consideration of a null arriving as the result of an adjacent LP sending a real message to some other LP. Regardless of whether LP x receives such a null message earlier than a periodic null, it still must wait for all events on adjacent LPs with times that fall between $e_{x,i-1}.t$ and $e_{x,i}.t$ to be processed. The waiting periods are simply shifted in simulation time.

When LP x sends a null message to LP z with time $\min(\mu(IC_x, i, g) + OC_x.d_z, e_{x,i}.t)$ for $1 \leq g \leq h_{x,i}$, there may be a number of events that LP z is able to process. When LP z receives this null, it processes up to the time allowed by the null. Thus, LP x must wait as long as it takes LP z to process those events before the next null is sent from LP z. LP z begins execution of $e_{z,j}$ and finishes with some other event, denoted $e_{z,j'}$. Thus, after the gth iteration of the input protocol for x, the time spent waiting for the next null message from a single LP $z \in ADJ(x)$, denoted $\tau(\omega(z, i, g))$, is:

$$\tau(\omega(z,i,g)) = \sum_{v=j}^{j'} (LP_z.\tau_{2,v} + LP_z.\tau_{3,v} + LP_z.\tau_{4,v}) \tag{21}$$

Each LP $z \in ADJ(x)$ may process a different number of events; each event may take different amounts of real time to process. Though individual null messages sent by LP x to adjacent processors must be captured separately by considering $|ADJ(x)|$, the time LP x spends waiting after sending the messages is bounded by the longest wait for any of the LPs, $\max_{\forall z \in ADJ(x)}(\tau(\omega(z, i, g)))$. For the interval $\Lambda(x, i)$, the wait can be expressed as:

$$\tau(\omega(ADJ(x,i))) = \sum_{g=1}^{h_{x,i}} \max_{\forall z \in ADJ(x)}(\tau(\omega(z,i,g))) \tag{22}$$

4.3.2.5 Maximum Number of Real Messages. The last term to be developed is an expression for the number of real messages that LP x can receive during $\Lambda_{x,i}$. The time a message is sent by LP z and the time it is to be executed on LP x are always identical in the current design. In order for LP z to send a real message with time $m.t_r$, its own clock must be set at that time. Since the clock value is equal to the time of the event currently being executed on LP z, and that event can only be executed if a null message with time $m.t_n \geq e_{z,j}.t$ was sent from LP x, it follows that any real message sent by LP z can be processed when z finally blocks. Thus, any real message from LP z to x ends the interval $\Lambda_{x,i}$ and begins $\Lambda_{x,i+1}$.

LP z may send more than one real message to LP x with the same simulation time. The actual number depends on implementation specifics as well as player crossings at that time in the simulation scenario. Scenario dependence makes reflection of a realistic upper or lower bound unlikely, so the quantity is modeled as an unspecified constant $\mathcal{M}_{x,i}$. When multiple events with the same simulation time, or simulation times in an interval less than $OC_z.d_x$, are sent from z to x, LP x will be able to process all the events sequentially with no additional null message processing or synchronization time. Thus, modeling the quantity as a simple unknown constant applied to message retrieval time is suitable.

4.3.3 Output Protocol Model. The output protocol occurs in two places: copy updating in step 4.1 and next event scheduling in step 4.3. In copy updating, copies may reside on different LPs. When this occurs, real messages with time set to the current simulation time are sent to transmit an update event to each LP containing a copy. Null messages with the same time are sent to adjacent LPs that do not contain a copy. When all player images reside in sectors on one LP, Equation 10 suffices for the parallel performance model. This corresponds to step 4.1.

Message transmission during event scheduling is similar to that during copy update, except that at most a single real message is sent at a particular simulation time, and only under certain circumstances. In particular, a real message is sent when a player copy in a sector on an LP other than the LP with the owning player image predicts an interactive event. This is the mechanism by which player visibility is maintained across LP boundaries. When the copy is in a sector on the same LP as the owning copy, the event is simply inserted in the NEQ. This corresponds to step 4.3.

Except for fine-grained decompositions in which many sectors or LPs are used, or pathological simulations in which players line up along LP boundaries and operate there continuously, the output protocol will be used infrequently and occur unpredictably. Thus, other portions of the algorithm will tend to dominate overall sequential and parallel execution time. Rather than burden the general model with complicated conditional expressions, the output protocol is represented as a single abstract term. However, a detailed expression is given for step 4 that can be used for analysis of special cases.

Copy update analysis is simplified by assuming that, from the time the front of a player contacts a boundary until the back of the player crosses the boundary, update events are generated by the primary, or *owner*, player image, transmitted as real messages to LPs with copy images, and scheduled on the NEQ in each receiving LP for subsequent execution. Further, each such message is tagged with the current simulation time. Thus, while copies of a player exist on multiple LPs for the duration of a crossing, each LP executes the same number, though not necessarily the same type, of events. This assumption is reasonable under the design criteria that all actions on one LP affecting objects on another LP be handled through the simulation mechanisms. In other words, if there are c copies, the owner sends c messages to effect an update. Under this assumption, the real message generated by a copy that goes to the owner to inform the owner of a required state update is captured as one of the c messages. This is suitable because the owner need not actually send a message back to the LP with that copy.

The time to insert an event in the NEQ at a receiving LP is modeled as part of the input protocol, as is the time the receiving LP might spend waiting for the message. Since the receiving LP does not know whether the next message it receives will be a real message or a null message, the model for waiting time need not distinguish between the two.

In the parallel model, the updating function is essentially the same, except that copy structures are distributed among LPs and the LP generating the update must also generate the requisite real and null messages. When the LPs reside on several processors, copy retrieval occurs in parallel rather than sequentially as on a single processor.

The cost of copy updating on LPs receiving update events is covered in event execution. For the generating LP, a distinction is needed for those player copies resident on the LP and those resident elsewhere. Let $L(C(p))$ be a function returning a subset of $C(p)$ for some player p such that the copies returned are resident on the local LP. Let $L' = C(p) - L$. The LP must update player images represented by L, and then distribute messages. This is expressed by modifying Equation 10 from the sequential model and specializing terms for both $e_i.p$ and $e_i.q$:

$$\tau_{4,i} \geq$$

$$\tau(v(L(C(e_i.p))) + (|L'(C(e_i.p))| \times \tau(\sigma(R))) + ((|OC_x| - |L(C(e_i.p))|) \times \tau(\sigma(N))) +$$

$$\tau(\delta(e_i.p)) + \tau(\iota_{n,i-1}) +$$

$$(e_i.q \neq \emptyset) \times (\tau(\upsilon(L(C(e_i.q))) + (|L'(C(e_i.q))| \times \tau(\sigma(R))) + \tau(\delta(e_i.q)) + \tau(\iota_{n,i}))) \quad (23)$$

This equation does not contain a representation for null messages when player q is updated, since that update happens at the same simulation time as the update for player p. Duplicate null messages need not be sent at the same simulation times.

4.3.4 Integrated Parallel Performance Model. Using limits for iterative execution of the input protocol, the time spent in the input protocol for the ith iteration of LP x, denoted $\tau_{1A,i}$, is:

$$\tau_{1A,i} = \mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_{n,i',g-1})) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x,i))) \quad (24)$$

This model is combined with models for other steps in the algorithm to produce a model for the finishing time of LP x running the parallel algorithm, τ_{par_x}:

$$\tau_{par_x} = \tau_1 + \sum_{i=1}^{s'+\mathcal{X}bx_w} (\tau_{1A,i} + \tau_{2,i} + \tau_{3,i} + \tau_{4,i}) \quad (25)$$

4.4 Summary

Models for sequential and parallel discrete event battlefield simulation algorithms developed in this chapter provide a basis for analysis of parallel performance. A method for estimating the number of boundary crossing events is shown for use in cost estimation of sectored decomposition. Use of the sequential model with sectoring incorporated allows more accuracy in comparisons between sequential and parallel performance by providing an abstract view of the better sequential approach. A model for the output portion of the protocol provides a detailed representation that can be used for special case analysis.

The models are used in Chapter V to examine performance factors theoretically for both the sequential and parallel algorithms. Model reduction is used to identify control parameters that may assist in improving the amount of concurrency extracted from a simulation. Empirical results are used to demonstrate potential improvements shown by model manipulation. Finally, several conclusions are drawn about potential algorithm improvements.

V. Model Analysis and Demonstration

This chapter contains manipulations and analysis of the models developed in Chapter IV to identify factors important to parallel performance. The manipulations focus on the effects of elements in the conservative synchronization algorithm. Measurements taken on the Intel iPSC/2 are used to demonstrate theoretical results obtained for special cases of model instantiation.

This chapter is presented in three sections. The first section contains the fully expanded models developed in Chapter IV and a series of model reductions to verify algebraically that input protocol behavior is isolated from simulation behavior when no real messages pass between LPs. This allows analysis of the minimum synchronization impact for any simulation. The second section outlines tests developed to demonstrate the analytic results and shows comparative execution time data collected during testing. The final section analyzes the results from the analytic and empirical studies in the context of the battlefield domain and provides several heuristics for parallel performance improvement and problem decomposition. The analysis includes identification of several design challenges identified in the modeling process.

5.1 Performance Models and Reduction

The complete finishing time models developed for a sequential simulation and a generalized LP in a parallel simulation are:

$$
\begin{aligned}
&\tau_{seq} = \\
&\quad \sum_{j=1}^{p} \Bigg[\sum_{d=1}^{c'} A(ec_d, p_j) \times \tau(\chi(ec_d, p_j)) + \\
&\quad \sum_{d=c'+1}^{c} A(ec_d, p_j) \times \Bigg[\sum_{p_k \in \alpha(p_j)} A(ec_d, p_k) \times \tau(\chi(ec_d, p_j, p_k)) \Bigg] + \tau(\iota_{n,j-1}) \Bigg] + \\
&\quad \sum_{i=1}^{s'+\mathcal{X}bx_w} \Bigg[\tau(\gamma_n) + \sum_{d=1}^{c} [(e_i \in ec_d) \times \tau(\eta(ec_d))] + \tau(\upsilon(C(e_i.p))) + \\
&\quad \sum_{d=1}^{c'} A(ec_d, e_i.p) \times \tau(\chi(ec_d, e_i.p)) + \\
&\quad \sum_{d=c'+1}^{c} A(ec_d, e_i.p) \times \Bigg[\sum_{p_k \in \alpha(p_j)} A(ec_d, p_k) \times \tau(\chi(ec_d, e_i.p, p_k)) \Bigg] + \tau(\iota_{n,i-1}) + \\
&\quad (e_i.q \neq \emptyset) \times (\tau(\upsilon(C(e_i.q))) + \sum_{d=1}^{c'} A(ec_d, e_i.q) \times \tau(\chi(ec_d, e_i.q)) + \\
&\quad \sum_{d=c'+1}^{c} A(ec_d, e_i.q) \times \Bigg[\sum_{p_k \in \alpha(e_i.q)} A(ec_d, p_k) \times \tau(\chi(ec_d, e_i.q, p_k)) \Bigg] + \tau(\iota_{n,i})) \Bigg] \qquad (26)
\end{aligned}
$$

and

$$
\begin{aligned}
&\tau_{par_x} = \\
&\quad \sum_{j=1}^{p} \Bigg[\sum_{d=1}^{c'} A(ec_d, p_j) \times \tau(\chi(ec_d, p_j)) + \\
&\quad \sum_{d=c'+1}^{c} A(ec_d, p_j) \times \Bigg[\sum_{p_k \in \alpha(p_j)} A(ec_d, p_k) \times \tau(\chi(ec_d, p_j, p_k)) \Bigg] + \tau(\iota_{n,j-1}) \Bigg] + \\
&\quad \sum_{i=1}^{s'+\mathcal{X}bx_w} [\mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_{n,i',g-1})) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x,i))) + \\
&\quad \tau(\gamma_n) + \sum_{d=1}^{c} [(e_i \in ec_d) \times \tau(\eta(ec_d))] + \tau(\upsilon(L(C(e_i.p)))) + (|L'(C(e_i.p))| \times \tau(\sigma(R))) + \\
&\quad ((|OC_x| - |L(C(e_i.p))|) \times \tau(\sigma(N))) + \sum_{d=1}^{c'} A(ec_d, p_j) \times \tau(\chi(ec_d, e_i.p)) +
\end{aligned}
$$

$$\sum_{d=c'+1}^{c} A(ec_d, e_i.p) \times \left[\sum_{p_k \in \alpha(e_i.p)} A(ec_d, p_k) \times \tau(\chi(ec_d, e_i.p, p_k)) \right] + \tau(\iota_{n,i-1}) +$$
$$(e_i.q \neq \emptyset) \times (\tau(v(L(C(e_i.q))) + (|L'(C(e_i.q))| \times \tau(\sigma(R))) +$$
$$\sum_{d=1}^{c'} A(ec_d, e_i.q) \times \tau(\chi(ec_d, e_i.q)) +$$
$$\sum_{d=c'+1}^{c} A(ec_d, e_i.q) \times \left[\sum_{p_k \in \alpha(e_i.q)} A(ec_d, p_k) \times \tau(\chi(ec_d, e_i.q, p_k)) \right] + \tau(\iota_{n,i}))) \Bigg] \qquad (27)$$

$OC_x.d_z, z \in ADJ(x)$, the output channel delay from LP x to each adjacent LP, is the single static control parameter in the parallel model. The parameter is embedded in the range for $h_{x,i}$ developed in Equation 20. The parameter is shown explicitly after several reduction steps since usage requires an assumption to instantiate either the upper or lower bound for $h_{x,i}$.

The number of sectors in the sequential model, the number of processors in the parallel model, and the particular configurations of sectoring strategies and mappings to processors are control parameters as well. The effects of the sectoring strategy are modeled by the dynamic behavior associated with α for each sector, the function that describes the number of players in the sector for each loop iteration. The effects of the sector-to-LP mapping, and thus of the processors used, build up from this as the number and kind of events that occur on an LP during a given time interval. The parallel performance model is expressed as the finishing time of a single LP. The LP with the largest number of events is chosen for model instantiation. However, no explicit combination of LPs and processors is considered at a lower level of detail.

General simulator parameters are c', the number of noninteractive event predictors and classes; c'', the number of interactive event predictors and classes; and A, the applicability function mapping event classes to player classes. The number of players, p, and the initial state of α for each sector are the scenario parameters.

In its general form, the sequential model could be used to calculate the finishing time of a sequential simulation by expanding each summation iteration and substituting measured or assumed values for each term. The parallel model could likewise be used by replicating it for each LP in a parallel simulation and expanding each term for each LP used. The maximum result over all the replications would be the predicted finishing time of the parallel simulation.

The primary interest of this study is the analysis of the conservative synchronization protocol on parallel simulation performance. To focus on this in the performance model, two sets of simplifying assumptions are made. The first set of six assumptions removes treatment of special cases and possible execution time variance from dynamic NEQ and player copy set lengths. Variance from different NEQ lengths between sequential and parallel implementations is also removed by the first set of assumptions.

The second set of four assumptions permits algebraic verification of input protocol isolation from simulation behavior for the perfectly parallel simulation in which no events on an LP cause events on other LPs.

5.1.1 Assumption Set 1.

1. All event classes apply to all player classes. Thus, all values of the applicability function A are 1. This eliminates event prediction variance due to selective applicability of event types to players.

2. The initial step for both algorithms is excluded. The time taken by this step is assumed to be small in comparison to the time taken by the main loop. Inclusion would bias performance results in favor of the parallel implementation since each LP works with a smaller portion of the total player complement than does the sequential implementation.

3. The main loop in the sequential model is indexed for iteration over S events. The main loop in the parallel model, which is expressed in terms of the work to be done by one of P LPs collectively executing S events, is indexed for iteration over S/P events. This assumes a uniform distribution of events per LP.

4. Event execution, prediction, and the input protocol are executed for every simulation loop. Thus, they are more likely to affect performance than sporadic activities such as copy updating and boundary crossing events. Model terms expressing copy updates and movement across sector and LP boundaries can be left in simple terms of the form $\tau(v(C(p)))$ and $\mathcal{M}_{i,x}$, rather than expanded to distinguish among numbers of copies, distribution of copies, and real messages passed among LPs.

5. NEQ size remains constant in a sequential simulation and on each LP in a parallel simulation. NEQ operations can be modeled as a constant, $\tau(\iota_n)$, rather than distinguishing NEQ sizes on each iteration.

6. Variance in NEQ size between a sequential and parallel implementations is not sufficient to cause significant variance in NEQ insertion time between the two versions.

With these assumptions, the sequential and parallel performance models become:

$$\begin{aligned}\tau_{seq} = {} & \\ & \sum_{i=1}^{S}\Bigg[\tau(\gamma_n) + \tau(\iota_n) + \tau(v(C(e_i.p))) + \sum_{d=1}^{c}(e_i \in ec_d) \times \tau(\eta(ec_d)) + \\ & \sum_{d=1}^{c'}\tau(\chi(ec_d, e_i.p)) + \sum_{d=c'+1}^{c}\sum_{p_k \in \alpha(e_i.p)}\tau(\chi(ec_d, e_i.p, p_k)) + \\ & (e_i.q \neq \emptyset) \times \Bigg[\tau(\iota_n) + \tau(v(C(e_i.q))) + \sum_{d=1}^{c'}\tau(\chi(ec_d, e_i.q)) + \\ & \sum_{d=c'+1}^{c}\sum_{p_k \in \alpha(e_i.q)}\tau(\chi(ec_d, e_i.q, p_k))\Bigg]\Bigg]\end{aligned} \qquad (28)$$

$$\begin{aligned}\tau_{par_x} = {} & \\ & \sum_{i=1}^{S/P}\Bigg[\tau(\gamma_n) + \tau(\iota_n) + \tau(v(C(e_i.p))) + \sum_{d=1}^{c}(e_i \in ec_d) \times \tau(\eta(ec_d)) + \\ & \sum_{d=1}^{c'}\tau(\chi(ec_d, e_i.p)) + \sum_{d=c'+1}^{c}\sum_{p_k \in \alpha(e_i.p)}\tau(\chi(ec_d, e_i.p, p_k)) + \\ & (e_i.q \neq \emptyset) \times \Bigg[\tau(\iota_n) + \tau(v(C(e_i.q))) + \sum_{d=1}^{c'}\tau(\chi(ec_d, e_i.q)) + \\ & \sum_{d=c'+1}^{c}\sum_{p_k \in \alpha(e_i.q)}\tau(\chi(ec_d, e_i.q, p_k))\Bigg] + \\ & \mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_n)) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x, i)))\Bigg]\end{aligned} \qquad (29)$$

The sequential model is now expressed in terms of insertions and removals in a single NEQ, player insertion, removals, and updates in multiple player sets, and event execution and prediction. The parallel model includes each of those features as well as event scheduling across multiple NEQ structures, protocol message processing, and time spent waiting on protocol messages. The perfectly parallel simulation of $K \times B$ players distributed uniformly over K sectors, where each sector is allocated to a single LP, is modeled with the remaining assumptions, 1–4. Since each sector is allocated to a single LP and each LP resides on a separate processor, $K = P$ where P is the number of processors used.

5.1.2 Assumption Set 2.

1. No interactive events are executed: $\forall i : e_i.q = \emptyset$.
2. No boundary crossings are executed, and thus no player copy updates are required. The number of players in a sector remains constant over time: $\forall i : C(e_i.p) = \emptyset$ and $|\alpha(e_i.p)| = B$.
3. The times to retrieve an event from the NEQ, insert an event in the NEQ, execute an event, predict a noninteractive event, and predict an interactive event are constants and interactive event predication takes V times as long as noninteractive event prediction: $\tau(\gamma_n) = C_0$; $\tau(\iota_n) = C_1$; $\forall d : \tau(\eta(ec_d)) = C_2$; $\forall i, d | 1 \leq d \leq c'$: $\tau(\chi(ec_d, e_i.p)) = C_3$; $\forall i, k, d | c' < d \leq c : \tau(\chi(ec_d, e_i.p, p_k)) = V \times C_3$.
4. The times to retrieve a message from a buffer and destroy a message are constants: $\tau(\gamma_b) = C_4$; $\tau(\zeta_b) = C_5$.

The finishing time of the parallel simulation, t_{par}, is equal to longest finishing time of any of the LPs used, max t_{par_x}.

By Assumption 1, the terms representing interactive event processing are eliminated, leaving:

$$\begin{aligned}
&\tau_{seq} = \\
&\sum_{i=1}^{S}\Bigg[\tau(\gamma_n) + \tau(\iota_n) + \tau(\upsilon(C(e_i.p))) + \sum_{d=1}^{c}(e_i \in ec_d) \times \tau(\eta(ec_d)) + \\
&\sum_{d=1}^{c'} \tau(\chi(ec_d, e_i.p)) + \sum_{d=c'+1}^{c} \sum_{p_k \in \alpha(e_i.p)} \tau(\chi(ec_d, e_i.p, p_k))\Bigg]
\end{aligned} \tag{30}$$

and

$$\begin{aligned}
&\tau_{par_x} = \\
&\sum_{i=1}^{S/P}\Bigg[\tau(\gamma_n) + \tau(\iota_n) + \tau(\upsilon(C(e_i.p))) + \sum_{d=1}^{c}(e_i \in ec_d) \times \tau(\eta(ec_d)) + \\
&\sum_{d=1}^{c'} \tau(\chi(ec_d, e_i.p)) + \sum_{d=c'+1}^{c} \sum_{p_k \in \alpha(e_i.p)} \tau(\chi(ec_d, e_i.p, p_k)) + \\
&\mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_n)) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x,i)))\Bigg]
\end{aligned} \tag{31}$$

By Assumption 2, no time is spent updating player copies. Further, the limit of summation for predicting interactive events is bound to B. This yields:

$$\begin{aligned}
&\tau_{seq} = \\
&\sum_{i=1}^{S}\Bigg[\tau(\gamma_n) + \tau(\iota_n) + \sum_{d=1}^{c}(e_i \in ec_d) \times \tau(\eta(ec_d)) + \\
&\sum_{d=1}^{c'} \tau(\chi(ec_d, e_i.p)) + \sum_{d=c'+1}^{c} \sum_{k=1}^{B} \tau(\chi(ec_d, e_i.p, p_k))\Bigg]
\end{aligned} \tag{32}$$

and

$$\tau_{par_x} =$$
$$\sum_{i=1}^{S/P}\Big[\tau(\gamma_n)+\tau(\iota_n)+\sum_{d=1}^{c}(e_i \in ec_d)\times\tau(\eta(ec_d))+$$
$$\sum_{d=1}^{c'}\tau(\chi(ec_d,e_i.p))+\sum_{d=c'+1}^{c}\sum_{k=1}^{B}\tau(\chi(ec_d,e_i.p,p_k))+$$
$$\mathcal{M}_{x,i}\times(\tau(\gamma_n)+\tau(\iota_n))+h_{x,i}\times(\tau(\gamma_b)+\tau(\zeta_b))+\tau(\omega(ADJ(x,i)))\Big] \quad (33)$$

In addition to simplifying other terms, substitution of appropriate constants from Assumption 3 permits resolution of the summation used to model mutually exclusive event class execution times:

$$\tau_{seq} =$$
$$\sum_{i=1}^{S}\left[C_0+C_1+C_2+\sum_{d=1}^{c'}C_3+\sum_{d=c'+1}^{c}\sum_{k=1}^{B}V\times C_3\right] \quad (34)$$

and

$$\tau_{par_x} =$$
$$\sum_{i=1}^{S/P}\Big[C_0+C_1+C_2+\sum_{d=1}^{c'}C_3+\sum_{d=c'+1}^{c}\sum_{k=1}^{B}V\times C_3+$$
$$\mathcal{M}_{x,i}\times(\tau(\gamma_n)+\tau(\iota_n))+h_{x,i}\times(\tau(\gamma_b)+\tau(\zeta_b))+\tau(\omega(ADJ(x,i)))\Big] \quad (35)$$

Simplification and regrouping of constants yields:

$$\tau_{seq} = S\times(C_0+C_1+C_2+c'\times C_3+(c-c')\times B\times V\times C_3) \quad (36)$$

and

$$\tau_{par_x} =$$

$$(S/P) \times (C_0 + C_1 + C_2 + c' \times C_3 + (c - c') \times B \times V \times C_3) + \sum_{i=1}^{S/P} [\mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_n)) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x,i)))] \quad (37)$$

Letting $C' = C_0 + C_1 + C_2 + c' \times C_3 + (c - c') \times B \times V \times C_3$ yields:

$$\tau_{seq} = S \times C' \quad (38)$$

and

$$\tau_{par_x} = (S/P) \times C' + \sum_{i=1}^{S/P} [\mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_n)) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \tau(\omega(ADJ(x,i)))] \quad (39)$$

The impacts and relationships of the parallel overhead can now be more carefully examined.

5.1.3 Parallel Overhead Model Reduction. Expanding the overhead portion using Equations 16 and 17 on page 62 yields:

$$\tau_{par_x} =$$

$$(S/P) \times C' + \sum_{i=1}^{S/P} \left[\mathcal{M}_{x,i} \times (\tau(\gamma_n) + \tau(\iota_n)) + h_{x,i} \times (\tau(\gamma_b) + \tau(\zeta_b)) + \sum_{g=1}^{h_{x,i}} \max \sum_{v=j}^{j'} (LP_z.\tau_{2,v} + LP_z.\tau_{3,v} + LP_z.\tau_{4,v})\right] \quad (40)$$

where $z \in ADJ(x)$, v denotes the next event for LP z such that $\mu(IC_x, i, g) \leq e_{z,v}.t \leq \mu(IC_x, i, g+1)$, j and j' are the first and last indices, respectively, of all such events executed by LP z, and $LP_z.\tau_{2,v} + \ldots + LP_z.\tau_{4,v}$ is the real time needed for LP z to execute event v.

Before further expanding $h_{x,i}$, several immediate simplifications can be made. Let $\mathcal{E}(e_{z,v}, g)$ denote the total processing time of the vth event on LP z. Then $\mathcal{E}(e_{z,v}, g) = LP_z.\tau_{2,v} + LP_z.\tau_{3,v} + LP_z.\tau_{4,v}$. Modeling the event processing time as a function of the event on LP z permitted to be processed by the gth protocol iteration on LP x addresses the fact that the gth iteration may produce a null message that increases $\mu(IC_z)$, possibly allowing 1 or more events to be processed on LP z. By Assumption 2, $\mathcal{M}_{x,i} = 0$. Finally, with Assumption 4, the reduction produces:

$$\tau_{par_x} = (S/P) \times C' + \sum_{i=1}^{S/P} \left[h_{x,i} \times (C_4 + C_5) + \sum_{g=1}^{h_{x,i}} \max \sum_{v=j}^{j'} \mathcal{E}(e_{z,v}, g) \right] \tag{41}$$

The upper bound for $h_{x,i}$ from Equation 19 can be used since, by Assumption 2, no boundary crossings are executed.

$$h_{x,i} \leq |ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \tag{42}$$

Setting the parameters involved by using this upper bound for $h_{x,i}$ yields:

$$\tau_{par_x} = \\ (S/P) \times C' + \sum_{i=1}^{S/P} \left[|ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times (C_4 + C_5) + \right. \\ \left. \sum_{g=1}^{|ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x}} \min \sum_{v=j}^{j'} \mathcal{E}(e_{z,v}, g) \right] \tag{43}$$

With the exception of the constants, none of the terms reduce further without making overly constraining assumptions.

5.1.4 Algorithm and Performance Model Review. The first term in Equation 43, $(S/P) \times C'$, represents the total time LP x spends processing events. The second term aggregates the total time spent sending null messages and waiting for null messages to be returned by adjacent LPs. While a precise measure of the run time for the LP must consider these terms iteratively over each event processed by LP x, consideration without the multiplier (S/P) permits analysis of the time spent processing null messages and waiting for null messages between each pair of events processed.

A single iteration of the summation term of Equation 43 models each iteration of the S/P iterations in the parallel simulation. The form of Equation 43 shows that the amount of time LP x spends waiting between productive computation cycles depends on two factors: the LP's own progress rate in covering the time interval between the last event and the next event to be executed, $\Lambda_{x,i}$, and the real time adjacent LPs $z \in ADJ(x)$ spend executing events in the same simulation time interval. A reduction in synchronization time requires reduction of one or both of these factors.

Reducing the time LP z spends executing events during $\Lambda_{x,i}$ reduces exploited concurrency over K LPs, since those events are the ones most likely to be able to be executed concurrently with events on LP x. The rate of progress through $\Lambda_{x,i}$ can be reduced only if the next event happens earlier or a larger channel delay amount is used. The former is problem-dependent and thus uncontrollable. The latter is controllable, but a brief analysis on page 103 shows how arbitrary channel delay increases can constrain the model fidelity of moving object simulations.

A review of the parallel algorithm model at step 1A.4 shows the dependency problem directly. LP x sends null messages to neighbors only when it can do nothing else. This forces LPs adjacent to x to wait when there may be an event that is safe to process. In fact, since every LP in any pair must wait on the other, a maximum speedup of $P/2$ is forced on even the most parallel simulation for any number of processors. A different approach may be indicated.

Placing null message transmission in step 1A.3 of the algorithm, prior to checking the NEQ for a safe event to process, allows adjacent LPs in any pair the opportunity to proceed even when each is processing an event. The change does not impact the ability to avoid deadlock, since LP x will still send null messages when it finally has no events safe to process.

The performance model changes in several ways. The parallel component model given in Equation 17 becomes:

$$LP_x.\tau_{1A,i,g,2} = \begin{cases} \tau(\sigma(N)) & \text{if } e_{x,i}.t \leq \mu(IC_x, i, g) \\ \tau(\sigma(N)) + V \times C_3 & \text{if } e_{x,i}.t > \mu(IC_x, i, g) \end{cases} \tag{44}$$

The time an LP spends per iteration waiting for a required null message is now bounded by a constant in the worst case, rather than a function of the progress of adjacent LPs. Continuing from Equation 43, the performance model using this algorithm modification becomes:

$$\tau_{par_x} =$$

$$(S/P) \times C' + \sum_{i=1}^{S/P} \left[|ADJ(x)| \times (1 + \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x}) \times (C_4 + C_5) + \tau(\sigma(N)) + V \times C_3 \right] \quad (45)$$

The additional null message received from each adjacent LP is grouped with $\frac{\Lambda_{x,i}}{OC_x.d_z+OC_z.d_x}$ null messages received from adjacent LPs. The new component for waiting time, $\tau(\sigma(N))+V \times C_3$, captures the null message sent for each event since it is included inside the summation. Since all terms inside the summation are constants, the final form is:

$$\tau_{par_x} =$$

$$(S/P) \times C' + (S/P) \times \frac{|ADJ(x)| \times \Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times (C_4 + C_5) + \tau(\sigma(N)) + V \times C_3 \quad (46)$$

Without the algorithm modification, the best case waiting time for LP x occurs when there is only one event in the NEQ on LP z with a time in the interval $\Lambda_{x,i}$. Returning to Equation 43 with this best case assumption, $j = j'$ so the equation can be rewritten as:

$$\tau_{par_x} =$$

$$(S/P) \times C' + \sum_{i=1}^{S/P} \left[|ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times (C_4 + C_5) + |ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times \mathcal{E}(e_{z,j}, g) \right] \quad (47)$$

which, when factored, yields:

$$\tau_{par_x} =$$
$$(S/P) \times C' +$$
$$\sum_{i=1}^{S/P} \left[|ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times (C_4 + C_5 + \mathcal{E}(e_{z,j}, g)) \right] \quad (48)$$

Taking the single event represented by $\mathcal{E}(e_{z,j}, g)$ to be executable in maximum constant time of $\tau(\sigma(N)) + V \times C_3$, Equation 48 reduces to:

$$\tau_{par_x} =$$
$$(S/P) \times C' +$$
$$(S/P) \times |ADJ(x)| \times \frac{\Lambda_{x,i}}{OC_x.d_z + OC_z.d_x} \times (C_4 + C_5) + \tau(\sigma(N)) + V \times C_3 \quad (49)$$

Equivalence of Equation 49, reflecting the best waiting time for LP x before the algorithm modification, with Equation 46, reflecting maximum waiting time for LP x after the modification, shows that the modified version never exhibits a null message wait longer than the shortest wait possible with the original algorithm. This holds as long as each LP in a pair has an event within the same simulation time interval, permitting concurrent event execution. The only cost is the additional null message generation. If the concurrently processed events take longer to execute than the null message takes to generate, the cost is absorbed. If either LP does not have an event in the interval, it wouldn't be processing anyway, so no penalty is incurred. As the number of events processed concurrently in the simulation increases, the relative expense in additional null message processing decreases. A similar result has been reported by Nicol (30:306).

5.2 Measurements and Demonstration of Model Results

This section describes measurements used to demonstrate model reduction results. The methods for measurement of basic simulation operations and specific time values are presented, as well as the details of simulation scenarios used to collect the measurement data. Basic operation time values were measured using a sequential simulation with uniform distributions of events over players with respect to simulation time; players over battlefield sectors in two dimensions; and, for parallel simulation operations and scenarios, uniform sector distribution over LPs and LPs over processors in one and two dimensions.

All measurements are taken on the Intel iPSC/2 with no other users or background tasks on the system.

5.2.1 Basic Operation Timing. The most significant operations in the simulation are event prediction, event execution, and NEQ insertion and removal. Several different scenarios with particular characteristics were used to collect average or minimum times for each operation. All scenarios were run on a single processor except for the scenario used to measure the time to generate and send a null message. This scenario was run on two processors.

Several of the basic operations take less than one millisecond (msec) to complete. Since the granularity of the system clock is one msec, these operations were timed by placing calls inside a loop executing 100, 1000, or 10000 times as appropriate to measure the operation. This method produces an average time measurement for the operation. Each of the loops was measured 30 times to gauge their effect on operation measurement. The 10000 cycle loop always takes either 13 or 14 msecs, while the 1000 cycle loop never exceeds 2 msecs. The average time for the 1000 cycle loop over 30 samples is 1.4 msec, consistent with expectations developed by the measurement for the 10000 cycle loop. The contribution of the 100 cycle loop is negligible. All times reported for operations timed over more than one iteration are adjusted to remove the effect of the loop.

Table 1 summarizes the minimum times measured for all BATTLESIM operations. Detailed tables in several subsections show specific behavior characteristics for certain operations based on numbers of players in sectors and ordering of data structures in the implementation.

Table 1. BATTLESIM Operation Time Measurements

Operation	Op Type	Time
Prediction $\tau(\chi)$	Route Point	496 μsec
	Collision	284 μsec
	Boundary	830 μsec
Execution $\tau(\eta)$	Route Point	710 μsec
	Collision	22 msec
	Boundary	4 msec
Null Msg $\tau(\sigma(N))$	Make and Send	1 msec

5.2.1.1 Event Prediction and Null Message Generation. The scenario used for timing event prediction included 20 players with 101 route points each. The scenario used a 3x3 grid of sectors to create one sector with four hard boundaries. Five players were placed in the center sector and one of the exterior sectors, and ten players were placed in one of the other exterior sectors. Players were overloaded in one sector to observe the degree of variation between collision prediction times in sectors with smaller and larger numbers of players. Players stayed within the sector in which they started. The scenario was configured to run with all sectors on a single processor. This scenario was also run on two processors to time null message generation. Two processors are sufficient to time the operation of generating and sending a null message.

Minimum times for event predictions collected over 1000 iterations are summarized in Table 1. These times are included with minimums for other BATTLESIM operations to provide relative comparisons.

As an interactive event class, collision prediction is affected by the number of players per sector. Table 2 shows average times for collision prediction collected for each player in the scenario. The times are grouped by sector and then by player reference within the sector. Prediction times for players in the second and third sectors used are greater than those for players in the first sector even though the first sector contained ten players while the second and third each contained five players. Each sector data structure is itself stored sequentially, with players referenced by the sector also stored sequentially. This leads to longer absolute prediction times due to a longer access time required to index further into the sector data structure. Within a sector, increasing times reflect the dependence of prediction time on the number of players in the sector.

In each set of times for a sector, the first time is substantially less than all following times. The first player reference is selected with a different function than all others, get_first vice get_next. The get_first function provides direct access to the first player, while the get_next function advances to the second player and iterates from that point. The same functions are used to retrieve the aggregate sector structures prior to referencing players within.

Table 2. Collision Prediction Times

Sector No.	Player in Sector	Time
1	1	284 μsec
	2	812 μsec
	3	825 μsec
	4	844 μsec
	5	860 μsec
	6	880 μsec
	7	894 μsec
	8	917 μsec
	9	930 μsec
	10	949 μsec
2	1	527 μsec
	2	1054 μsec
	3	1069 μsec
	4	1087 μsec
	5	1104 μsec
3	1	651 μsec
	2	1175 μsec
	3	1194 μsec
	4	1211 μsec
	5	1229 μsec

5.2.1.2 Event Execution. Three scenarios were used to time event execution. Route point events were timed using the scenario with 20 players allocated to four sectors. Collision execution was timed using a scenario with 60 players colliding with one another in pairs. This scenario used one sector on one processor. Finally, boundary crossing event times were collected using a scenario with 30 players crossing between two sectors. The players were divided into two waves following one after the other. This provided collision avoidance while keeping the scenario manageable.

Table 1 includes the lowest average times for execution of each event type. The time for route point execution is stable regardless of the number of players in the sector. The time is taken using 100 iterations of the event for each occurrence. The lowest time for collision execution occurs when the last pair of players in the sector structure collide. Collision execution times are high enough to omit multiple iterations for measurement. Times shown for boundary event execution in Table 1 are simple averages. While the times were sufficient to omit multiple iterations, they varied significantly from player to player. This is attributable to a number of conditional evaluations exercised in each boundary event execution.

Times for each collision in the 30-collision scenario are shown in Table 3. Values in the table are listed from left to right, top to bottom in the order in which the collisions occurred. Three of the player pairs apparently disappear from the simulation without colliding. Cursory attempts to ascertain why this happens were unsuccessful.

Collision execution time generally decreases as players are removed from the scenario. However, some times increase between collisions, reflecting the variations from colliding players being referenced in different areas of the sequential data structure in the sector. No attempt was made to have players collide in a particular order.

Table 3. Individual Collision Execution Times (in msec)

68	55	71	58	50	64	54	46	59
49	42	54	45	38	49	40	34	38
31	34	28	32	25	28	23	25	22

5.2.1.3 NEQ Operations. Table 4 shows average times to insert and remove an event for several NEQ lengths, corresponding to model parameters $\tau(\iota_n)$ and $\tau(\gamma_n)$. These times demonstrate the contribution to simulation loop execution time added by the NEQ for various numbers of players per LP. These measurements represent the average time needed to insert an element at the end of a list of the given length and remove the front element. Since insertion and removal occurs for every event executed, the two operations were measured as an aggregate over 1000 samples.

Table 4. NEQ Operation Measurements

NEQ Length	Time
1	13 msec
10	14 msec
100	18 msec
1000	60 msec
8000	394 msec

5.2.2 Simulation Experiments. The reduction of potential LP waiting time described on page 83 was demonstrated using a scenario of 20 players. The route for each player consisted of 101 points, resulting in 100 events per player. Each route formed a straight line and did not intersect with any other route. All routes were designed to generate an event for each player at every time unit. The resulting simulation is balanced with respect to event frequency per player and execution time per event.

For run time measurement, timing began with the first event to be executed and ended with the last event executed. This removed the overhead associated with file input and the initial round of event prediction and scheduling.

The first set of comparative measurements used the synchronization algorithm as shown in Figure 6 on page 56. The second set of comparative measurements used the algorithm with the modification proposed in Section 5.1.4 on page 83. In both sets of measurements, the scenario was run first on a single processor using a single sector. Another sequential test was run using 4 sectors, each with 5 players. Player routes did not cross any sector boundaries. Finally, each set of measurements concluded with parallel tests in which each sector was allocated to a processor. All player routes remained the same as in the sequential simulation.

The sequential tests were run using a single LP representation. This adds a constant amount of overhead per event execution since several function calls are made to determine whether another LP has sent a message. The function calls would not be made in a strict sequential simulation that does not implement the LP model. The overhead added per event executed is approximately 60 μsec, so it is not significant. The parallel tests were run using 4x1 and 2x2 processor configurations. Each configuration was run 10 times to collect an average run time. The averages from each of these tests are shown in Table 5. There was no time variance among the individual trials.

Table 5. Synchronization Algorithm Improvement Results

Test Set 1	Sequential	1 sector	78 secs
		4 sectors	60 secs
	Parallel	4x1	30 secs
		2x2	30 secs
Test Set 2	Sequential	1 sector	78 secs
		4 sectors	60 secs
	Parallel	4x1	17 secs
		2x2	17 secs

These results demonstrate the potential of using null messages to increase concurrent execution when available in the simulation scenario. For sequential processing, the results demonstrate the potential savings available from sectoring when sector crossing overhead is not sufficient to overshadow time savings from reducing player references in event prediction.

5.3 Analysis

Conservative synchronization has been shown to be capable of recognizing differing amounts of parallelism in different problem models (30:323). The goal of this work is to identify factors and relationships that determine performance in parallel battlefield simulation. This goal defines the context for analysis.

5.3.1 Parallel Performance Issues.

5.3.1.1 Control Parameters. Equation 46 shows that channel delay time, $OC_x.d_z$ for an LP x with adjacent LP z, is the only control parameter that affects the behavior of the input protocol. The general sequential and parallel performance models, Equations 5.1 and 5.1 respectively, show that the sectoring strategy and sector mapping to parallel processors remain as the only additional control parameters.

The sectoring strategy and processor mapping determine player relationships to one another and to player groupings, as well as the maximum number of events that can be processed at a given time. However, without explicit knowledge of player distributions in space over time, the opportunity to control these parameters to improve performance is limited.

To a large extent, the benefits and costs of sectoring and processor allocation vary with player movement during each event iteration. Sectoring within the boundaries of a single processor may reduce player references in event prediction, but it also adds the potential for generating events that are not present in the sequential simulation. When the time spent executing these events exceeds the time saved in player references, the sectoring scheme hurts performance. Since LP boundaries coincide with sector boundaries, the implication for parallel processing is that the time spent processing boundary crossings should not exceed the gains made from processing events concurrently. Concurrent processing gains can only be made when players on different LPs have events that can be executed during the same simulation time frame, and only then when the synchronization algorithm can recognize the concurrency and permit multiple event execution.

Control of the channel delay increment with respect to the difference between the last event and next event to be processed is a key factor in recognizing concurrency. Identification of unnecessary sequencing among processors shown on page 83 is the first step in exploiting inherent concurrency. The next step is the reduction of protocol messages each LP generates. Protocol messages add time to the processing of event schedules on both the sending and receiving LPs. Real time is added by message production or consumption and waiting for protocol messages with timestamps that permit processing of the next event.

By Equation 19, the number of null messages between two LPs x and z per event processing iteration is inversely proportional to the total channel delay time between the two LPs. Equally important, the timestamp change between successive messages is proportional to delay time. Thus, increasing the delay time reduces messages produced and consumed and increases the rate of progress of LPs throught their event schedules. The delay time controls several rates of change in the algorithm and across the LP network.

1. *Progress of LP x.* The immediate cycle formed between every LP and its $|ADJ(x)|$ neighbors has the effect of injecting $|ADJ(x)|$ null message transmissions and receipts, at a minimum, between every event processed by LP x. This effect can be

limited by restricting the conditions under which null messages are sent, as in the canonical algorithm, but at the expense of also limiting opportunities for concurrent execution. The rate of progress of a single LP through its schedule is a function of the ratio between its inter-event times and its output delay.

2. *Progress of LP x with respect to LP $z \in ADJ(x)$.* The rate of progress of one LP through its event schedule establishes a maximum rate of progress for all LPs in its adjacency set. The amount of concurrent event execution is determined by the inter-event times among LPs in an adjacency relationship. This measure is only indirectly related to the rate of progress of any one LP. Even with a slow rate of advance, events can still be executed concurrently; the periods between concurrent execution are simply longer.

3. *Progress of LP x with respect to arbitrary LP $l \notin ADJ(x)$.* Since each LP in a mesh topology is directly dependent on neighbors and indirectly dependent on all other LPs, the primary determinant of progress for a particular LP is the tightness of binding in simulation time delay between adjacent LPs. As this is relieved, LPs further away in the topology become more binding on progress. When considered from the point of view of absolute simulation time, the difference between lowest NEQ time and highest $\mu(IC_x)$ establishes a window of concurrency, $\mathcal{W}$, across all LP schedules. All events in the window can be executed concurrently. The window's rate of movement across the set of LP schedules is an aggregate measure of the 'speed' of the simulation.

5.3.1.2 Inherent Concurrency. Performance assessment of a parallel algorithm must be based on the amount of concurrency in the problem solved by the algorithm. Data decomposition assumes that there is some amount of data independence that can be used to allow parallel execution. If problem data is completely independent, decomposition over p processors can result in p-fold speedup. As dependencies are added, the amount of concurrency available in the problem decreases. In simulation, dependencies are added only by changing the event trace, thus changing the simulation itself. The amount of *inherent concurrency*, and thus performance potential, differs widely among possible simulations regardless of the synchronization algorithm used.

In a deterministic simulation model, the amount of concurrency is embedded in the initial state of the simulation. The simulator acts as a generator, producing an event trace according to the rules of the simulation model. The parallel simulator uses multiple processors that each begin with a portion of the initial state and execute the simulation model rules sequentially.

Few metrics have been proposed to quantify the amount of concurrency inherent in a problem as a function of time or event ordering. The *Event Horizon*, adapted from physics, has recently been offered as an analytic model of inherent concurrency (49:39). This appears to be the best metric yet described for establishing a baseline for synchronization algorithm performance. Its application in battlefield simulation has been tested with promising results, but specific use in algorithm construction is not yet widespread (50).

Figures 7 and 8 represent event traces with different amounts of available concurrency. Each square represents an event in a process schedule. Each event is defined to require one unit of real processing time. Events occur at integral values of simulation time as shown inside the squares.

The absence of inter-schedule arrows in Figure 7 indicates that no event dependencies exist among schedules. This simulation could be run with no synchronization at all. However, since this is not known beforehand, synchronization must be used. Synchronization based on simulation time alone, using a channel delay of one unit between all processors, forces nine iterations to process the total schedule when it could be done in four with four processors. Though the scenario has a high degree of concurrency, conservative synchronization using next event times and fixed channel delays does not fully recognize it.

Arrows in Figure 8 show event dependencies across schedules. This figure represents a simulation with some inherent parallelism. However, conservative synchronization using event time stamps alone would not recognize most of the concurrency. In this example, four events are scheduled on each of four processors by some method associating events and processors, such as spatial decomposition in battlefield simulation. Time-stamp ordering forces sixteen iterations, exactly the minimum number needed to process the total schedule sequentially.

5.3.1.3 Concurrency Recognition.

In Figures 7 and 8, each event occurs at uniform time intervals and is defined to require a unit amount of time to execute. When

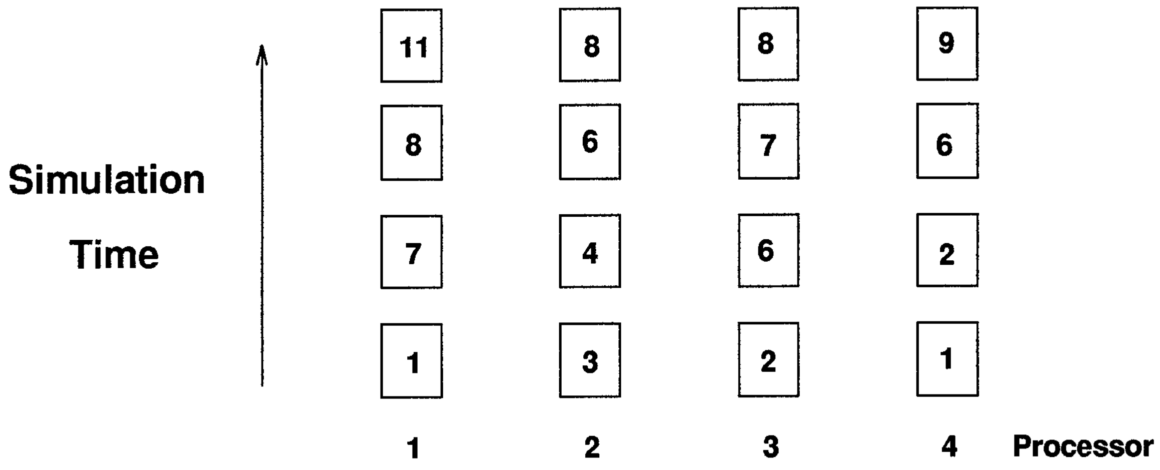

Figure 7. Completely Independent Event Schedules

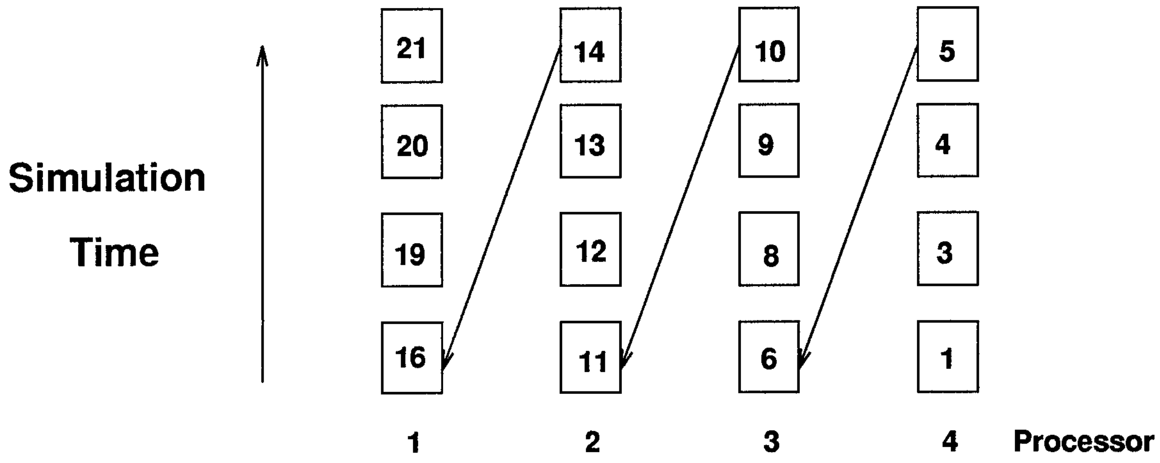

Figure 8. Dependent Event Schedules

similar conditions do not hold for event time distribution and real time execution duration, they both become performance factors.

When events in the simulated system occur aperiodically, the number of increments each LP takes to reach its next event increases over what would occur for simulation of a synchronous real system. The reduced model on page 85 shows this explicitly, since the number of null messages produced and received increases as the time between events on an LP increases. Each null message between events constitutes an unnecessary synchronization attempt. When events take different amounts of real time to execute, processes arrive at logical synchronization points at different points in time, thus forcing dependent processes to wait to receive protocol messages. The ability of the synchronization algorithm to advance LPs quickly toward required synchronization points indicates its ability to recognize concurrency in the problem.

In a battlefield context, the required synchronization points correspond to players crossing LP boundaries. Inside each of the intervals between boundary crossings, there is no need for processor interaction. Thus, each processor should be permitted to run every event it generates up to the time of the next actual boundary crossing into or out of the area covered by the processor. This suggests three possibilities for improved concurrency recognition:

- Earliest possible LP boundary crossing times can be used to determine potential synchronization points. After each event executed, or less frequently for efficiency, the earliest crossing that coincides with an LP boundary can be computed using player distances and maximum velocities. The time can be broadcast globally using the interconnection network or distributed using null messages. As a global synchronization, channel time management and error reporting under this scheme would have to be relaxed so that null messages communicating the global minimum time would not be considered to be out of time order.
- An LP may be able to use the simulation event structure to deduce limits of possibility for incoming or outgoing events based on current state. The event structure can be provided to the LP as an abstract tree or graph to preserve decoupling in the

general simulation architecture. The graph models the hierarchy of events and event production rules used by the simulator to model player behavior.

- The Event Horizon concept could be used to develop a risk-free synchronization algorithm suitable for battlefield simulations.

Each of these options may provide a performance improvement over the current method for determining future earliest message times. The null message improvement, similar in nature to the conditional event synchronization method (8), may provide reasonable savings for the work expended. Abstract event structure specification and interface design would be somewhat more complicated, but would provide the synchronization protocol with information more closely resembling the rules used to generate events in the first place. Finally, use of the Event Horizon to construct a risk-free synchronization algorithm (49:41) may provide the most payoff in the broadest variety of scenarios.

Sector Crossing Times. For each event executed, a player's state, including position, is updated. This may determine a new earliest possible time in the future for an LP boundary crossing. Thus, the earliest possible intersection of all player routes with LP borders can be computed and substituted for the NEQ time component used in determining the time on the next null message sent to that LP. Tables 3 and 4 show that border intersection calculations do not take significantly longer than collision prediction, and are currently overshadowed by the NEQ operation times.

Minor modifications to this algorithm can be made and the resulting information provided to the synchronization protocol for use in null message generation. An adjustment has to be made when the player actually hits the border, since the computation at that point would provide a zero time advance. Some small absolute minimum could be substituted.

This approach would bring all currently available problem information to bear on moving LPs forward in simulation time as fast as possible. When the difference between the current time and the next possible crossing time is greater than the advance provided by the current strategy, the approach will allow a larger simulation time window for concurrent event execution. This type of approach amounts to an improvement in *lookahead.* The absence of lookahead has been shown to be detrimental to performance in conservative protocol applications (17:24).

Event Structure As a Parameter. An immediate weakness of using event times as a synchronization device is that they only partially describe the range of possible next events and next event times. Thus, if this is the only parameter available to the synchronization protocol, the ability of an LP to determine its future may be overly limited.

Event structure analysis has been proposed for various purposes in PDES, including automatic lookahead computation (10) and simulation speedup estimation (47). This idea could be extended to the design of a generic interface that allows simulation event generation and precedence information to be passed to the synchronization protocol during simulation initialization. The protocol logic could then use the structural information, combined with time knowledge present during each state, to further limit and possibly more accurately deduce next event time minimums passed to adjacent LPs.

The success of this approach is certainly dependent on the degree of detail specified in the event structure. However, as the modeling details of the simulation application are developed to finer levels of granularity, this approach would seem to offer some potential in closing the gap between the concurrency recognition provided by event time data and actual problem concurrency.

Event Horizon. Intuitively, the best synchronization algorithm would deliver each process precisely to its next synchronization point without any delays along the way. The Event Horizon concept appears to provide a theoretical approach to doing this. In addition to serving as a metric for gauging inherent concurrency, the concept could be used for synchronization algorithm construction.

5.3.1.4 Channel Delay and Replicated Players. The channel delay used by an LP x to allow adjacent LPs to progress signifies the earliest time in the future that LP x will send another message. If a vehicle of size s traveling through LP x at velocity v contacts the boundary with another LP z at time t, LP x sends a message at time t to LP z to create a copy of the vehicle in LP z. A similar action occurs at time $t + (s/2v)$ when the center of the vehicle hits the boundary. This second message causes LP z to generate a null message back to LP x with time $t + (s/2v) + OC_z.d_x$, allowing LP x to process events up to that time. However, if the vehicle continues at its current direction and velocity, its back will cross the boundary at time $t + s/v$. LP z will send a message to LP x at this time to report the crossing. If $(s/2v) < OC_z.d_x$, a causality error occurs. Preventing the error requires either a redesign of the data replication approach or a requirement that $OC_z.d_x < (s/2v)$. For large v and small s, such as those characteristics exhibited by aircraft, $OC_z.d_x < (s/2v)$ compromises performance by inducing a large number of null messages for each event processed. The relationship between the size of $OC_z.d_x$ and time between events to parallel performance is established in Equation 46.

In addition, LP x will process all events in its current permissible time interval before an adjacent LP z can process events. If an interaction between players on adjacent LPs occurs within this time interval, it cannot be correctly modeled by the simulation. When LP z resumes processing, its simulation time will be less than that of LP x. Thus, if LP z detects and attempts to schedule a shared event with LP x at a time during LP z's current processing interval, the event will necessarily be in error since it will be in the simulation past for LP x.

This behavior was first observed during routine use of BATTLESIM. To isolate the problem and explore a solution, a scenario was designed in which a single player crosses a sector boundary coincident with an LP boundary between two sectors on two processors. Several runs were made, varying the velocity and size of the player so that $(s/2v) < OC_z.d_x$ was both preserved and violated. When the constraint is preserved, the synchronization protocol performs without incident. When it is violated, the algorithm consistently reports a message ordering error, indicating a causality constraint violation.

To address this issue, a special function was developed to alter the channel delay time when a player crosses a boundary. When the front end of the player hits the boundary, its maximum velocity and size are used to determine the center crossing time with respect to the current time. A special null message is sent around the cycle between the two involved LPs to set the channel clock times appropriately, since the previous null message sets them too far into the future. The new minimum increment is used until the back of the player crosses into the gaining sector. At that point, another special null message is used to reset the delay times to the previous value.

This approach was tested using the same scenarios that caused error reporting before the approach was implemented. No errors were reported, and no opportunity for real causality errors was introduced.

5.3.2 General Performance Issues.

5.3.2.1 Sectoring Benefits and Costs.

Sectoring a sequential simulation saves time if the number of player references needed for event prediction is reduced over a non-sectored implementation and new events corresponding to sector crossings do not outweigh the reduction (53:916) (31:187). The intuition behind this approach is apparent and was demonstrated in model reduction and testing. Table 1 shows measurements indicative of this savings. However, the net benefit is affected by movement and feature interactions that vary unpredictably over time. Both the savings and cost vary with interaction and player movement.

Boundary event prediction takes as much time as the other two measured predictors taken together. Boundary event execution is of moderate cost with respect to other event execution times. The cost of interactive event prediction is driven by the insertion and reference time requirements of the data structures used to hold players in sectors. Reduction of these times depends on either reducing the size of the data structure between index points or reducing the overall complexity of data structure manipulations.

In the battlefield simulation application, the size between index points changes with player movement on each iteration. While development of ways to predict player flow into and out of the structures may prove useful in the long term, complexity reduction through the use of more efficient data structures such as hash tables is likely to provide the easiest observable improvement.

The dimensionality of sectoring is another consideration. For sequential simulation with no knowledge of future motion, two-dimensional sectoring provides the least interior boundary distance for a given number of sectors. This produces the smallest opportunity for boundary crossings to be generated, and thus the lowest cost potential. For parallel simulation with unknown motion characteristics, two dimensional decomposition provides the highest possibility of concurrent execution. Decomposition in a single dimension forces sequential execution of events for players that are in the same dimensional strip, even though they may be very far away from one another. For most cases, two dimensional decomposition for a given number of sectors should yield the best results.

5.3.2.2 Player Replication. In sequential processing, the player copy management scheme used to ensure proper player event prediction and state update increases overhead by increasing the number of player images referenced in event prediction. This type of overhead is similar in sequential simulations and SPMD simulations in which a player's prediction range spans sectors but remains confined to a single processor.

When a player's prediction range spans processor boundaries in SPMD simulations, the approach introduces an additional effect on top of the iterative complexity. Since player state data is replicated in the sectors and associated processors into which the player has visibility, state data update and event processing must occur in strict timestamp order on all affected processors. This amounts to a strict time order dependency among the processors, removing the possibility of parallel execution while the data is replicated.

Simple player crossings of joint processor/sector boundaries at high speed present the least potential impact to parallel execution due to the player management scheme. The greatest potential impact occurs when a player moves parallel to a boundary while straddling it. Parallel execution that could occur otherwise is nullified, and protocol messages are generated at a rate of one in each direction between processors per minimum time interval.

The impacts to parallel performance are a function of the movement of the player and the time spent predicting events for the player over the other players in the sectors in which it is replicated. Thus, the impacts are inherently unpredictable from the structure of the simulated space unless significant constraints on detection range are levied. Even with such limitations, the dynamic effects of event prediction are still present.

5.3.2.3 NEQ Performance. Time measurements in Table 1 show that NEQ insertion time is a dominant factor in the sequential portion of the algorithm. This result underscores the need to use efficient data structure implementations regardless of whether the simulation is implemented sequentially or in parallel. The NEQ is a heavily used data structure. From a practical point of view, efficient NEQ operation stands to reduce expected run time substantially in any implementation. From a comparative perspective, NEQ distribution over a distributed processor network will necessarily result in better performance, and thus clouds any comparisons that might be made between parallel and sequential implementations.

5.4 Summary

This chapter presents theoretical and empirical results that identify factors important to parallel performance in battlefield simulation. Model analysis and test results provide insight into improvements that can be made in conservative parallel synchronization algorithms, as well as inherent limitations. Data structures used to maintain event lists and player collections are identified by timing measurements as driving factors in simulation loop iteration time. Hash table implementation is identified as a design approach that would make data structure performance less sensitive to simulation dynamics. Reduction of data structure complexity plays an important role in assessing true parallel algorithm performance, since inefficiencies in sequential data management obscure algorithmic performance differences attributable solely to parallel implementation.

VI. Summary of Results and Recommendations

6.1 Summary of Results

The objective of this research was to develop analytic performance models of sequential and parallel discrete event battlefield simulation using conservative synchronization for use in identification of factors that affect sequential and parallel performance.

The objective was met. A general deterministic performance model was developed for both sequential and parallel battlefield simulations in terms of finishing time. The model structure was specialized and reduced to explore expected behavior attributable to certain algorithm components. Model reduction that focused on exploring the static contribution of the parallel synchronization algorithm revealed a design anomaly that limits potential speedup by a factor of two. Empirical results confirmed this behavior for a perfectly balanced simulation scenario. Model analysis also provided insight into potential solution approaches. The synchronization design was modified as suggested by the analysis and speedup within a constant of linearity was demonstrated for simulations with a high degree of inherent concurrency. The potential for this type of result was not clear prior to model development.

The modeling approach used was sufficient to capture relationships among static synchronization algorithm components and the battlefield simulation model that had the potential to limit exploitation of potential concurrency. In addition to supporting the research objective, the approach was also sufficient to suggest directions for improvement using domain-independent and domain-specific information.

The predictive quality of the model was sufficient to show expected orders of magnitudes of difference in finishing times between sequential and parallel simulation implementations for test cases in which the degree of inherent concurrency was established. While prediction of absolute finishing times was not a specific objective of the research, the utility of developing the performance model in those terms was clearly demonstrated for comparative purposes.

An indirect result of this work was the identification of inherent conflict between the Chandy-Misra null message deadlock avoidance scheme and the current data replication management approach used in BATTLESIM. While this is primarily an implementation detail, it does show a need for better integration between the replication manager and the synchronization protocol for effective simulation of objects that span a large geographic region.

6.2 Research Contribution

This work contributed to current general knowledge in two ways. It demonstrated the effectiveness of using deterministic modeling techniques to explore parallel performance improvements in dynamic simulation models. It also provided a theoretical model for use in evaluating the potential of conservative synchronization protocols to provide good parallel performance in battlefield and moving object simulations.

In addition, this research provided a number of insights useful to the continuing exploration of parallel discrete event battlefield simulation at the Air Force Institute of Technology. The models developed can be used to focus on particular areas of algorithm design that indicate tight coupling which forces unnecessary synchronization. They can be refined to support study of potential heuristics that use domain-specific information, such as the inclusion of combat operational doctrine and mission planning data, to improve the performance of parallel battlefield simulations.

6.3 Recommendations for Further Study

6.3.1 General Directions. Fujimoto outlines four proposals to make the benefits of "...parallel simulation more accessible to the simulation community" (18). The essence of three of the proposals is to remove the burden of dealing with parallelism from the application. Nonetheless, his assessment of the state of PDES research progress finds that the best results have been obtained in cases in which application programmers were experts with the synchronization algorithm, while the worst results have been cases in which application programmers had limited experience with it (18:220). This implies that a non-trivial coupling must exist between the application and the synchronization algorithm if good performance is expected.

The general parallel simulation architecture, of which the testbed at the Air Force Insitute of Technology is an example, supports information hiding and design modularity. The next step should be to design a general interface that supports the transfer of lookahead and event precedence information from the application down through each level while maintaining semantic abstraction.

The simulation kernel needs access to event precedence information for proper event scheduling and cancellation. The synchronization protocol needs abstract future event time information to exploit concurrency as it unfolds in the simulation. The protocol could also benefit from abstract knowledge of event precedence, since the structure defines the rules by which events can be generated. Parameterized transfer of a precedence graph representation could be used to maintain decoupling.

Several research efforts have examined formalizations of event precedence structure and their use in synchronization (10) (47). Formalizations of inherent concurrency have been somewhat more elusive. However, Steinman recently advanced the "Event Horizon" as a formal metric of inherent simulation concurrency (49). It appears to apply to any simulation model, and provides the insight needed to construct a general synchronization algorithm that performs well regardless of the target application. His own research has included implementation of the concept and produced excellent results in parallel battlefield simulations using conservative synchronization (50). Future work should focus on incorporating this notion into parallel simulation algorithm design.

6.3.2 Specific Issues. Several short-term areas for investigation are indicated, involving issues in both parallel and sequential modeling and design. An overarching recommendation for exploring any of these areas is that they be tackled individually rather than all at once. A measured approach is likely to produce more readily verifiable results than attempting to solve all of the issues in concert.

- *Performance Model Validation.* The models resulting from this research were developed using constructive analysis. Empirical studies were limited to exploration of a very small set of model parameters. While the constructive process and details are thoroughly described herein, complete validation and expansion of abstract terms would provide a more complete and usable performance model repertoire for the various components of the battlefield simulation design.

- *Next Event Queue Design.* Timing experiments confirmed that operations on sequential event list structures can quickly dominate computational complexity. While this is neither new nor a particular impediment to this research, it does highlight the need for incorporation of more efficient data structures if credible demonstration of traditional speedup results becomes a research objective. This observation applies to both the event list and data structures used for object collections. Operations on both types of structures can obscure synchronization algorithm behavior. Structures and techniques demonstrated to be efficient in other efforts, such as the SPEEDES queue (49:40) and hash table implementations (50:9), may be beneficial.

- *Domain-Specific Information.* Generally, parallel simulation research efforts reported in the literature have not indicated the use of actual combat model information in the exploration of conservative synchronization protocols. State-of-the-art research in simulation of dynamic systems often uses statistical data collected from the domain of interest. Advanced work in theoretical performance modeling could benefit from the use of this type of information, both in terms of finely tuned modeling and academic credibility. As part of the Department of Defense infrastructure, AFIT is uniquely positioned to take advantage of this type of information.

- *Object Replication Scheme.* The reliance of the current object replication scheme on simultaneous events across logical processors fundamentally conflicts with the general deadlock avoidance strategy. This was demonstrated both analytically and empirically. Synchronization protocol implementation improvements developed as part of associated research may help to resolve the problem. However, future work dealing with moving object simulations should look at this area in more detail, especially in the context of performance degradation.

- *Event Precedence.* In addition to exploring the abstract event precedence specification for performance improvement in the long term, it should be considered in the short term as a way to resolve the occurrence of simultaneous events across logical process boundaries. The event structure implementation built explicitly in the testbed software could be used to formulate a more abstract design.

6.4 Summary

This chapter summarizes the overall research effort in the scope of the initial objectives, methods, resulting model products, and demonstrative empirical results. The chapter also describes recommendations for future research based on observations made during model development and demonstration. The chapter concludes with recommendations for long- and short-term research objectives for both sequential and parallel simulation component performance evaluation.

Appendix A. Data Dictionary and Function Definitions

This appendix details the entire complement of data definitions, abstract data types, and abstract function definitions used in the development of performance models for parallel discrete event battlefield simulation. Data model interpretations and structural constraints on data definitions establish a context for model usage consistent with the moving object problem domain. Definitions and functions described with the performance models are replicated here for completeness.

A.1 Data Dictionary

The data dictionary is composed of formal definitions, informal descriptions, structural constraints, and data model interpretations.

A.1.1 Definitions and Descriptions.

- $TIMES = \{x | x \text{ is a simulation time}\}$

  $TIMES$ is the set of simulation times projected or used in a simulation.

- $SIM\_ENT = \{x | x \text{ is a simulated entity}\}$

  SIM_ENT is the set of types of physical components or systems that are simulated.

- $ATT = \{x | x \text{ is an attribute of a simulated entity}\}$

  ATT is the set of attributes that comprise the state space of instances of simulated entities.

- $PLYR = \{\langle z, ATT_z \rangle | z \in \mathcal{Z}^+ \wedge ATT_z \subset \mathcal{P}(ATT) - \{\emptyset\}\}$

$PLYR$ is the set of instances of simulated entities, hereafter called *players*. As a model of a physical component or system, each player has a *unique* identifier drawn from the positive integers and a single, non-null set of attributes. Attribute set commonalities may exist among players, providing the basis for player classes. Each player has an associated type that is the first attribute in the attribute tuple. Each player's attribute set remains static throughout its existence, though attribute values may change. The value of the type attribute for a given player remains constant during the player's existence.

- $BHVR = \{x | x \text{ is a simulated behavior}\}$

  $BHVR$ is the set of behaviors or processes that operate on physical components or systems modeled in the simulation. The number of behaviors modeled in a simulation must be finite. Members of $BHVR$ are modeled by iterative computational processes.

- $INTACT = \{TIMES \times (BHVR \cup \{\emptyset\}) \times (PLYR \cup \{\emptyset\})^2\}$

  $INTACT$ is the set of all possible applications of behaviors or processes that can be applied to up to two players at any time.

- $e \overset{\text{def}}{=} \langle t, b, p, q \rangle | \langle t, b, p, q \rangle \in TIMES \times (BHVR \cup \{\emptyset\}) \times (PLYR \cup \{\emptyset\})^2$

  e is the general form of a particular event in a DES.

- $EC_x = \{e \in INTACT | e.b = x\}$

  EC_x is the set of all events that model a behavior or physical process. Each set corresponds to a block in a partition of $INTACT$ and is known as an *event class.* The value of x distinguishing an event class remains constant throughout a simulation.

- $EC = \{EC_x | x \in BHVR\}$

EC is the set of all distinct event classes in a simulation.

- $PC_x = \{\langle i, ATT_i\rangle \in PLYR | (ATT_{i,1} = x)\}$

  PC_x is the set of all players sharing a value for the type attribute. Each PC_x is known as a *player class* and corresponds to a block in a partition of $PLYR$. The value of x distinguishing a particular player class remains constant throughout a simulation.

- $PC = \{PC_x | x \in SIM\_ENT\}$

  PC is the set of all player classes distinguished by a type attribute value.

- $IS\_A_{EC_x} : INTACT \rightarrow EC_x$

- $IS\_A_{PC_x} : PLYR \rightarrow PC_x$

  $IS\_A_{EC_x,PC_x}$ are functions mapping events and players to event and player classes, respectively. The functions are alternative definitions of EC_x and PC_x for given x.

- $A' : EC \times PC \rightarrow \{0, 1\}$

  A' defines the association between event classes and player classes for a simulation:

$$A'(ec_i, pc_j) = \begin{cases} 1 & \text{if } ec_i \text{ applies to } pc_j \\ 0 & \text{otherwise} \end{cases}$$

- $A : EC \times PLYR \rightarrow \{0, 1\}$

  A defines the association between event classes and individual players in a simulation:

$$A(ec_i, p_j) = \begin{cases} 1 & \text{if } A'(ec_i, pc_k) = 1 \wedge p_j \in pc_k \\ 0 & \text{otherwise} \end{cases}$$

- $INTER = \{\langle t, b, p, q\rangle | \langle t, b, p, q\rangle \in INTACT \wedge p, q \neq \emptyset\}$

$INTER$ is the set of all interactive events in a simulation that operate on exactly two players. Event classes in a simulation may be subsets of $INTER$.

- $IND = \{\langle t,b,p,q\rangle | \langle t,b,p,q\rangle \in INTACT \wedge q \neq \emptyset\}$

  IND is the set of all events that operate on exactly individual player. Event classes in a simulation may be subsets of IND.

- $CNT = \{\langle t,b,p,q\rangle | \langle t,b,p,q\rangle \in INTACT \wedge p, q = \emptyset\}$

  CNT is the set of all events that are used to control aspects of the simulation but do not directly operate on any players.

- $SIM = \{\langle e_i\rangle | \langle e_i\rangle$ is an event and $1 \leq i \leq \mathcal{N}$ where $\mathcal{N}$ is a fixed free integer$\}$

  SIM is the ordered set of event instances in the simulation of interest.

- $PTS = \langle x,y\rangle | x,y \in \mathcal{R}^+$

  PTS is the set of all possible points in two-dimensional space.

- $SECTS = \{\langle i,l,u\rangle | i \in \mathcal{Z}^+, l,u \in PTS\}$

  $SECTS$ is the set of physically disjoint but border-contiguous sectors, or regions, into which a battlefield is divided. Each sector is uniquely described by an integer index and both lower and upper two-dimensional, *min* and *max*. References to $r \in SECTS$ make use of only the first component of the tuple, the sector number, unless explicitly noted otherwise.

- $IN : PLYR \rightarrow SECTS$

IN is a function mapping a player, either owner or copy, to the sector in which it currently resides. $|IN| = p$; $IN^{-1}(x)$ is the set of players in sector x; $|IN^{-1}(x)|$ is the number of players in sector x.

- $C : PLYR \rightarrow P'|P' \in \mathcal{P}(PLYR)$

  C is a function mapping a player to copies of itself resident in other sectors. Elements of C are created when the front of a player crosses a sector boundary and are removed when the player's back crosses the same boundary.

- $m \overset{\text{def}}{=} \langle t, f, A\rangle | \langle t, f, A\rangle \in TIMES \times \{R\} \cup \{N\} \times \mathcal{P}(ATT)$

  m is the general form of a message in a PDES. t is the time of the message. f is the message type, either R or N for *Real* and *Null.* Real messages convey state data passed from one LP to another, while null messages distribute clock data among processors.

- $MSGS = \{m | m \text{ is a message}\}$

  $MSGS$ is the set of all messages generated between processors in a parallel simulation.

- $ch \overset{\text{def}}{=} \langle s, d, t, m\rangle | \langle s, d, t, m\rangle \in LP^2 \times TIMES \times MSGS \cup \{\emptyset\} \times \wedge m \neq \emptyset \Rightarrow t = m.t$

  ch is the general form of a communication channel between two LPs in a PDES where s is the source LP, d is the destination LP, and t is the current channel time. The channel time is set to be the time of the message currently in transit on the channel. If there is no message in transit, the channel time remains set to the time of the previous message.

- $CH = \{ch | ch \text{ is a channel}\}$

CH is the set of channels defined for a particular set of LPs.

- $CH_x = \{ch | ch \in CH \land x \in LP\}$

  CH_x is the set of channels defined for LP x. $CH_x \subseteq CH$.

- $IC_x = \{ch | ch \in CH_x \land x \in LP \land ch.d = x\}$

  IC_x is the set of all input channels defined for LP x.

- $OC_x = \{\langle ch, d\rangle | ch \in CH_x \land x \in LP \land ch.s = x \land d \in TIMES\}$

  OC_x is the set of all output channels defined for LP x. Each output channel has an associated delay time, d. The designator is distinct from the d destination attribute present at the base class level. The destination attribute is never used explicitly in subsequent modeling.

- $LP = \{\langle x, n, t, b, C\rangle | x$ is a logical process $\land n$ is a NEQ $\land t \in TIMES \land b$ is a buffer $\land$ $C = CH_x\}$

  This model uses a specialization of the general LP model (27:51) described in Chapter 2. In addition to sequential code, message handling capability, and communication channels, each LP has a NEQ, a clock, and a message buffer. The NEQ and buffer are described in Section 4.1.3. The sequential code corresponds to a particular simulation and (possibly null) interprocess synchronization algorithm. The message buffer contains all messages received by the LP that have not yet been processed.

- $ADJ : LP \rightarrow \mathcal{P}(LP)$

ADJ is a function that maps an LP to its adjacent LPs. In this model, all LPs are adjacent to at least one other LP. Adjacent LPs have both an input and output channel going in each direction. This determines causal dependence.

A.1.2 Structural Constraints.

1. **TIMES.** $TIMES \subset \mathcal{R}^+$
2. **PLYR.** $PLYR$ must be a function, though it is neither surjective or injective.

   $\forall p | p \in PLYR \Rightarrow \exists i, c | i \in IS\_A_{PC_x} \wedge i = \langle p, c \rangle \wedge c \in PC$
3. **Disjoint classes.** All classes at the same level of abstraction (PC_x and EC_x; $INTER, IND$, and CNT) are disjoint.

A.1.3 Data Model Interpretation. A meaningful universe of discourse for each of the sets is assumed. In addition to the structural constraints, several other restrictions are applied in consideration of the performance models.

1. The simulator terminates by scheduling and eventually executing an **End** event, a member of CNT. The simulator does not modify itself.
2. Player creation and destruction during the simulation does not affect termination.
3. Class membership, as well as membership in either of the IS_A relations, varies with creation or destruction of members rather than change in the distinguishing characteristic of the class.

4. Event classes are an abstraction independent of the existence of particular event instances. Derived event class memberships in $INTER$, IND, CNT, and A do not vary during a simulation.

5. Motion and locations are considered over one or two dimensions (1D or 2D) as specified in context.

6. Sector divisions are uniform in either dimension.

A.2 Performance Model Definitions and Descriptions

The performance models express the behavior of the canonical algorithms quantitatively by reference to data elements, relation membership, and algorithm structure. Set and relation membership and cardinality are the primary vehicles for translation between the abstract collective view found in the data definitions and the iterative, elemental view needed in the performance model. Additional definitions used in the performance models are cardinality expressions and abstract computational functions describing the work outlined in the algorithm. Abstract data types and functions are used to model major simulation data components.

A.2.1 Cardinality and Arithmetic Definitions. All cardinalities are parameters taken from the simulation of interest. With the exception of the number of events in the simulation, s', all are observable or measurable at some point prior to completion of the simulation. As a practical matter, the value of s' is not known prior to termination of simulation represented by SIM.

- $c' = |IND|$.

- $c'' = |INTER|$.
- $c = c' + c'' = |EC|$.
- $p = |PLYR|$.
- $s' = |SIM|$.
- $r' = |SECTS|$.

A.2.2 Reference Conventions. Individual elements of sets needed for explicit reference in the performance model are appended with appropriate subscripts resolved by indices of summation.

- p_i refers to the ith player $p \in PLYR$.
- ec_i refers to the ith event class $ec \in EC$.
- r_i refers to the ith sector $r \in SECTS$.
- e_i refers to the ith event $e \in SIM$.
- $\tau(op)$ refers to the real time needed to perform operation op.
- $\tau_{s,i}$ refers to the real time needed to perform the ith iteration of step s where the step is referenced to an algorithm model. Steps not falling in loops are referenced as τ_s.
- LP_x refers to the logical process labeled x.
- EC ordering. Noninteractive event classes are ordered so as to precede interactive event classes. Special class ordering is unspecified.

A.2.3 Abstract Data Types and Function Definitions. The primary abstract data types used in the models are the NEQ and message buffers. Every LP has one of each. A NEQ is a generic priority queue with defined but unspecified operations. Each of the operations completes in finite, measurable time and thus can be provided as an argument to an appropriate measurement function. Message buffers are used in the parallel algorithm. A buffer is essentially identical to a NEQ except that it holds queued interprocessor messages rather than events. Each LP in a parallel simulation has a message buffer that holds messages coming from the input channels associated with the LP. Messages are maintained in a buffer in nondecreasing order by time.

Several functions are defined for both data types. Functions subscripted with n operate on a NEQ and events, while functions subscripted with b operate on a buffer and messages.

- ι_n, ι_b - Insert item into structure. This representation does not explicitly model the operating efficiency of the structure. Dependency on the progress of the simulation is introduced when needed by use of additional subscripts.
- γ_n, γ_b - Remove and return the first item from structure.
- ζ_n, ζ_b - Delete item from structure.

A.2.4 Operation Definitions. An untyped function, $\tau(x)$, represents the real simulator time or work to do x. Typed functions model particular primitive operations of interest in the canonical algorithms. In each definition, $p_x \in PLYR \wedge e_i \in SIM$.

- $\chi(ec_i, p_j)$ and $\chi(ec_i, p_j, p_k)$ - Calculate next instance of event class ec_i for player p_j or for player p_j with respect to player p_k for $p_j, p_k \in PLYR$.
- $\delta(p_i)$ - Determine next event for player p_i.
- $\eta(ec_j)$ - Execute an event e_i in class ec_j for players $e_i.p$ or $e_i.p$ and $e_i.q$ as specified. This is an event class-wide worst case execution.
- $v(P\_SET), P\_SET \subset PLYR$ - Update the players in P_SET. Used to update the copies of a player to maintain consistency with respect to simulation time.
- $\mu(C\_SET, i, g), C\_SET \subset CH$ - Return the minimum time of the channels in C_SET with respect to loop i and protocol iteration g.
- $\sigma(N), \sigma(R)$ - Send a real or null message between two processors.
- $\alpha(p_i) = IN^{-1}(IN(p_i))$ - Shorthand notation to construct the set of players, including copies, in the same sector with player p_i.
- $\omega(ADJ(x), i, g)$ - Denotes LP x waiting on reception of null messages from its adjacent LPs as part of the gth input protocol iteration before processing the ith event.

A.3 Summary

This appendix enumerates the complete data dictionary and symbol legend used to construct the performance models for sequential and parallel battlefield simulation. Several additional symbols are introduced in the text during performance model construction. However, their meanings are evident in context. They arise primarily as a result of model manipulation and simplification.

Appendix B. The General Task Allocation and Scheduling Problem

This appendix summarizes several heuristic approaches to the general task scheduling problem for parallel algorithms. These approaches are formulated in the context of task dependencies and control flow parallelism, rather than data dependencies and parallelism used in battlefield PDES decomposition. However, at the next higher level of abstraction, the two types of problems are very similar. Thus, these examples serve to show generals ways in which the problem can be solved, the work needed to achieve these results, and the performance benefits that might be expected.

B.1 Task Allocation and Scheduling Approaches

The result of decomposing a problem with respect to inherent parallelism is a set of tasks to be scheduled on available processors. Typically, the goal of scheduling is to make an assignment of tasks to processors that will achieve the shortest elapsed execution time (14:8).

Each task in the set is often related by precedence of execution with respect to other tasks in the set. If there are no such relationships, scheduling is known as task allocation. A partial precedence among tasks may result in the ability to compute an optimal schedule in polynomial time. The result may also be that computation of an optimal schedule is NP-complete (14:8).

The computational complexity of the general scheduling problem has given rise to a number of heuristic methods that attempt to provide good solutions with minimal constraints. The most fundamental of these have recently been summarized (14). Others are variations that can generally be characterized as methods of "graph reduction, preemptive scheduling, max-flow min-cut, domain decomposition, and priority list scheduling" (44:223). Most are based on some form of graph or list representation of the problem and make use of a particular hardware architecture to simplify experimentation. Usefulness of the methods is measured in terms of their generality; the quality of decomposition and mapping they provide; and their own computational complexity. The following techniques for decomposition and mapping are representative.

B.1.1 Nearest-Neighbor (NN) and Recursive Clustering (RC). Iterative parallel programs can be modeled with an accurate Task Interaction Graph (TIG). This model captures tasks as nodes for which weights are assigned based on the computational complexity of each task. Edges between nodes are undirected and indicate a communication requirement between their incident nodes. Edge weights describe the relative cost of communication. For an iteration, all tasks can proceed independently but must synchronize to exchange results. The NN and RC mapping approaches both attempt to balance load and minimize communication costs. NN addresses load balance explicitly and communication cost implicitly, while RC inverts these emphases (40:2–3).

The NN strategy first groups tasks into clusters and assigns the clusters to processors while maintaining the nearest-neighbor property. The first mapping is modified iteratively using boundary refinement in an attempt to improve processor loading within the constraints of the NN property. Conversely, the RC strategy starts with all tasks in one cluster and proceeds to break this down recursively so as to minimize the total weight of inter-cluster edges. Within this constraint, the load in each cluster is kept as nearly equal to the load in other clusters as possible. The total number of clusters produced is equal to the total number of available processors. The second step maps the resulting clusters to processors so as to minimize total interprocessor communication (40:5–6,9).

Either method can be applied to any TIG. Experiments tracing the mapping time for both methods show that random TIGs favor RC when message setup time is not considered. Analysis using a representative cost model predicts that the NN strategy will yield better speedup for hypercubes with high message setup times and programs with a large amount of spatial locality, while RC is better suited for programs represented by random TIGS or running on hypercubes with low message setup times (40:13–15).

B.1.2 Heavy Node First (HNF) and Weighted Length (WL). HNF and WL are priority list scheduling methods based on algorithms that can be represented by a Directed Acyclic Graph (DAG). A DAG is similar to a TIG in that the nodes of the graph represent tasks in the problem and the weight of each node corresponds to the execution time of the associated task. Unlike a TIG, the edges of a DAG are directed and represent dependencies among tasks rather than simple interactions.

In HNF, tasks are grouped first by level and then sorted within each level by precedence and complexity. Each task ready to be processed is placed in the current level; that is, all tasks superior in precedence to the target task must have completed. This constraint ensures preservation of task dependencies. If the target task has no superiors, this condition is trivially satisfied. Within a level, the next processor to be assigned a task is that processor which has the lowest scheduled time thus far as determined by the weights of tasks already assigned. In general, there will be more tasks in a given level than available processors. Choosing the next task scheduled to be the one with the largest weight ensures an equitable distribution of total load among processors. Dummy nodes with weight calculated to balance concurrent tasks are used to fill holes left when the current level contains fewer tasks than available processors (44:224-5).

The WL algorithm is intended to be an improvement of the traditional Critical Path Method (CPM). In CPM, the length of the critical path for each node is computed using the node weights. As in HNF, the next processor to be scheduled is that which has the shortest total accumulated weight yet scheduled. Of the tasks ready for processing, the one with the longest critical path length is selected for the next processor. Dummy weighted nodes are used to fill holes when unscheduled nodes remain but none are currently ready for processing. Since CPM allows scheduling of a heavy node concurrently with possibly lighter dependent nodes, the method can result in a number of processors idling awaiting for completion of the heavy node. WL allows consideration of the weighted length of all children of all currently contending nodes. Rather than selecting the node with the longest remaining path as in CPM, WL selects the node with greatest weighted length. In the worst case, the schedule generated by WL will be the same as that generated by CPM. However, consideration of the weighted length in a manner similar to HNF provides a better schedule in most cases (44:225-9).

While the algorithmic complexity of WL is shown to be greater than that of HNF, experimental results indicate that for all DAGs except their respective worst cases, the two algorithms tend to produce schedules of similar quality. Since the worst case, especially that for WL, is unlikely to occur in practice, this implies that the extra effort needed to develop and build a WL implementation in the context of a real application may not be justifiable. Further, the applicability of WL is limited by its need for information about the entire DAG. Thus, problems represented by partitioned DAGs are not amenable to solution by WL (44:231).

B.2 Summary

A number of heuristic approaches have been developed for the general task scheduling problem. Each of these approaches uses application-specific data to make choices, either statically or dynamically, to control parallel program execution.

In battlefield PDES with spatial decomposition, similar problems exist. The number of application variables that change with respect to simulation time and player movement far surpasses the simple application models used to describe these algorithms for control-flow parallel processing. Nonetheless, these examples provide useful insight into the effects of unknown data dependencies and processor load imbalance caused by spatial decomposition can affect overall parallel performance.

Appendix C. AFIT PDES Research

AFIT PDES research activities have included investigations of battlefield, aircraft system, queuing system, and circuit design simulation; synchronization protocol behavior; use of software engineering design techniques; and graphic interface development. This appendix summarizes those efforts to present a rough chronology of progress to date as a platform for considering future work.

C.1 AFIT PDES Testbed

The primary DES environments available for use at AFIT are battlefield simulation (BATTLESIM and Parallel Ada Simulation Environment (PASE)), Very High Speed Integrated Circuit (VHSIC) Hardware Description Language (VHDL) simulation (VSIM), and queuing model simulation (QUEUESIM). These environments have evolved over the course of the AFIT parallel simulation research program to employ parallelism using the Chandy-Misra conservative synchronization approach as implemented in the SPECTRUM host manager. Researchers have investigated a variety of problems using these environments for validation.

Rizza developed the first sequential battlefield simulation system at AFIT in 1990. His research focused on modeling collisions between objects. To this end, the first generation of BATTLESIM kept track of object position and velocity in a master list, producing collisions when two objects moved to the same position at the same time. This work provided a starting point for AFIT DES research (37).

In 1991, Moser developed a parallel simulation of colliding pool balls. He used a single dimension spatial decomposition to allocate regions of the table to processors. Each processor was responsible for maintaining position and velocity information for the balls in its region, as well as for communicating state information to adjacent processors when balls crossed area boundaries (28). This approach is similar strip techniques analyzed in the literature (20:315).

Soderholm's work in 1991 parallelized the discrete event battlefield simulation of Rizza using conditional event execution (8), a performance economization within the conservative paradigm; and a variation of optimistic local rollback (13). Speedup measurements on a distributed memory machine showed experimental evidence that battlefield PDES execution could benefit from a hybrid synchronization approach. The work also provided empirical evidence that expected speedup varies directly with event interleaving (46).

Bergman combined the results of Rizza and Moser in 1992 to produce the next generation of BATTLESIM, a parallel discrete event simulation (PDES). Bergman used a two dimension, fixed boundary decomposition corresponding to battlefield grid sectors, with each square assigned to a logical process and subsequent one-to-one process-to-processor assignment. Simulation players were handled by the processor associated with the battlefield square in which they were located. Bergman also increased the amount of object interaction by adding a projecting sensor to each object, similar to radar in aircraft, which allowed objects to detect and react to the presence of other objects at a defined distance (4).

Looney demonstrated the feasibility of incorporating state retention and rollback with conservative synchronization to support interactive user commands for graphic display. His focus was on showing that the simulation could be halted, backed up to a previous point in time, and restarted from that point without corrupting simulation results (26). Looney's work combines concepts of conservative and local optimistic synchronization, supporting the premise that synchronization is more appropriately envisioned as a continuum rather than as two discrete paradigms (35).

C.2 Object-Oriented Parallel Simulation

Booth's research in 1991 focused on mapping the Software Engineering Institute's (SEI) model for Object-Oriented Design (OOD) and simulation onto parallel architectures. The paradigm is related to the OCU model developed by SEI and currently used as the basis for DoD simulation development. Booth used an existing SEI implementation of a Direct Current Electrical System simulation, written in Ada, as the basis for parallelization in Ada. Booth's primary emphasis was on showing the propriety of OOD in the design of parallel simulations (5).

In 1993, Trachsel further refined BATTLESIM to incorporate object-oriented design principles. He designed and implemented an object model that allows for straightforward incorporation of new players and environment objects. Trachsel maintained Bergman's partitioning scheme and converted a portion of the existing code use object-oriented programming techniques in C (51).

Belford examined the benefits of using object-oriented methods directly from the beginning in his design of a Parallel Ada Simulation System. Belford used Rumbaugh's object-oriented analysis and design methods (39) to construct the environment at a high level. His prototype in Classic Ada used a simulation mechanism and battlefield application similar to that found in BATTLESIM. For synchronization, Belford stayed with the Chandy-Misra protocol. Belford's effort established a point of departure for investigating simulation executive components that conform to recent DoD standards for modeling and simulation (2).

C.3 Parallel Simulation Task Scheduling Research

In 1989, Huson examined empirical approaches to static decomposition and dynamic load rebalancing in parallel discrete time simulations. Using a commercially developed, configurable Ballistic Missile Defense simulation, Huson laid out a set of guidelines for both static decomposition and dynamic rebalancing (23).

Sartor's investigation in 1991 focused on task scheduling algorithm analysis in VHDL simulation. The simulation is viewed as a constrained iterative task system. A polynomial time scheduling algorithm, the level strategy, is proposed and validated both theoretically and empirically. The level strategy minimizes latency, the time between successive iterations of a given task. This result is amplified with theoretical formulation of upper and lower bounds on latency for tasks of fixed and variable execution time (41).

Van Horn's work in 1992 compared four synchronization protocols based on algorithms proposed by Reynolds and Chandy and Misra. These comparisons used Reynolds' SPECTRUM filter with modifications to allow for easy protocol modification. Experimentation centered on a queueing model simulation of a car wash. Van Horn began the research with empirical guidelines postulated on his analysis of schematic graphs representing various simulation and filter configurations. He concluded with final guidelines adjusted for a variety of decomposition and synchronization schemes across several Intel iPSC/2 node configurations (52).

In 1993, Kapp examined the decomposition problem as it relates to VHDL simulation. He proposed a method for measuring the cost of a decomposition developed using a variety of strategies. His results validated the ability of the cost function to capture relative differences among decompositions in the VHDL simulation environment (24).

Bibliography

1. Adve, Vikram S. and Mary K. Vernon. "The Influence of Random Delays on Parallel Execution Times," *Performance Evaluation Review*, *21*(1):61–73 (June 1993).

2. Belford, James T. *Object-Oriented Design and Implementation of a Parallel Ada Simulation System*. MS thesis, AFIT/GCE/ENG/93D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1993.

3. Berger, Marsha J. and Shahid H. Bokhari. "A Partitioning Strategy for Nonuniform Problems on Multiprocessors," *IEEE Transactions on Computers*, *C-36*(5):570–80 (May 1987).

4. Bergman, Kenneth C. *Spatial Partitioning of a Battlefield Parallel Discrete Event Simulation*. MS thesis, AFIT/GCS/ENG/92D-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1992. AD-A258911.

5. Booth, Guy R. *Implementation of an Object-Oriented Flight Simulator D. C. Electrical System on a Hypercube Architecture*. MS thesis, AFIT/GCE/ENS/91D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1991. AD-A243700.

6. Breeden, Thomas A. *Parallel Simulation of Structural VHDL Circuits on Intel Hypercubes*. MS thesis, AFIT/GCE/ENG/92D-01, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1992. AD-A258999.

7. Chandy, K. M. and J. Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Software Engineering*, *SE-5*(5):440–52 (September 1979).

8. Chandy, K. M. and R. Sherman. "The Conditional Event Approach to Distributed Simulation." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by Brian Unger and Richard Fujimoto. 93–9. P.O. Box 17900 San Diego CA: Simulation Councils Inc., March 1989.

9. Chou, Chien-Chun and others. "A Generalized Hold Model." *Proceedings of the 1993 Winter Simulation Conference*, edited by Gerald W. Evans and others. 756–61. December 1993.

10. Cota, Bruce A. and Robert G. Sargent. "A Framework for Automatic Lookahead Computation in Conservative Distributed Simulations." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by David Nicol. 56–9. PO Box 17900 San Diego CA: Simulation Councils Inc, January 1990.

11. Davis, Nathaniel and others. "Distributed Discrete-Event Simulation Using Null Message Algorithms on Hypercube Architectures," *Journal of Parallel and Distributed Computing*, *8*(4):349–57 (April 1990).

12. Deo, Narsingh and others. "Processor Allocation in Parallel Battlefield Simulation." *Proceedings of the 1992 Winter Simulation Conference*, edited by James J. Swain and others. 718–25. December 1992.

13. Dickens, Phillip M. and Paul F. Reynolds. "SRADS With Local Rollback." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by David Nicol. 161–4. PO Box 17900 San Diego CA: Simulation Councils Inc., January 1990.

14. El-Rewini, Hesham and others. *Task Scheduling in Parallel and Distributed Systems*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1994.

15. Ferrari, Domenico. *Computer Systems Performance Evaluation*. Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1978.

16. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by Brian Unger and David Jefferson. 14–20. PO Box 17900 San Diego CA: Simulation Councils Inc., February 1988.

17. Fujimoto, Richard M. "Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*, edited by Edward A. MacNair and others. 19–29. December 1989.

18. Fujimoto, Richard M. "Parallel Discrete Event Simulation: Will the Field Survive?," *ORSA Journal on Computing*, *5*(3):213–30 (Summer 1993).

19. Greenbaum, Anne. "Synchronization Costs on Multiprocessors," *Parallel Computing*, *10*(1):3–14 (March 1989).

20. Hanxleden, Reinhard and L. Ridgway Scott. "Load Balancing on Message Passing Architectures," *Journal of Parallel and Distributed Computing*, *13*(3):312–24 (November 1991).

21. Hartrum, Thomas C. *AFIT Guide to SPECTRUM*. Department of Electrical and Computer Engineering, School of Engineering, Air Force Institute of Technology, Wright-Patterson AFB, OH 45433, September 1993.

22. Hoare, C. A. R. *Communicating Sequential Processes*. Englewood Cliffs, New Jersey 07632: Prentice-Hall International, 1985.

23. Huson, Mark Leslie. *An Empirical Development of Parallelization Guidelines for Time-Driven Simulation*. MS thesis, AFIT/GCS/ENG/89D-10, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1989. AD-A215665.

24. Kapp, Kevin L. *Partitioning Structural VHDL Circuits for Parallel Execution on Hypercubes*. MS thesis, AFIT/GCE/ENG/93D-07, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1993. AD-A274390.

25. Kumar, Vipin and others. *Introduction to Parallel Computing - Design and Analysis of Algorithms*. Redwood City, California 94065: The Benjamin/Cummings Publishing Company, Inc., 1994.

26. Looney, Douglas Clifford. *Interactive Control of a Parallel Simulation From a Remote Graphics Workstation.* MS thesis, AFIT/GCE/ENG/93D-9, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1993. AD-A274217.

27. Misra, Jayadev. "Distributed Discrete-Event Simulation," *Computing Surveys, 18*(1):39–65 (March 1986).

28. Moser, Robert S. *A Spatially Partitioned Parallel Simulation of Colliding Objects.* MS thesis, AFIT/GCS/ENG/91D-15, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1991. AD-A243967.

29. Nicol, David M. "Mapping a Battlefield Simulation onto Message-Passing Parallel Architectures." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by Brian Unger and David Jefferson. 141–6. P.O. Box 17900 San Diego CA: Simulation Councils Inc., February 1988.

30. Nicol, David M. "The Cost of Conservative Synchronization in Parallel Discrete Event Simulation," *Journal of the Association for Computing Machinery, 40*(2):304–33 (April 1993).

31. Nicol, David M. and Scott E. Riffe. "A Conservative Approach to Parallelizing the Sharks' World Simulation." *Proceedings of the 1990 Winter Simulation Conference*, edited by Osman Balci and others. 186–90. December 1990.

32. Nutt, Gary J. "Distributed Simulation Design Alternatives." *Proceedings of the SCS Multiconference on Distributed Simulation*, edited by David Nicol. 51–5. P.O. Box 17900 San Diego CA: Simulation Councils Inc., January 1990.

33. Quinn, Michael J. *Designing Efficient Algorithms for Parallel Computers.* New York, New York 10020: McGraw-Hill, 1987.

34. Reynolds, P. F. and P. M. Dickens. "SPECTRUM: A Parallel Simulation Testbed." *Proceedings of 4th Conference on Hypercubes, Concurrent Computers, and Applications.* 865–70. P.O. Box 428 Los Altos CA: Golden Gate Enterprises, March 1989.

35. Reynolds, Paul F. "A Spectrum of Options for Parallel Simulation Protocols." *Proceedings of the ACM Winter Simulation Conference.* 671–9. December 1988.

36. Rich, David O. and Randy E. Michelsen. "An Assessment of the MODSIM/TWOS Parallel Simulation Environment." *Proceedings of the 1991 Winter Simulation Conference*, edited by Barry L. Nelson and others. 509–18. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, December 1991.

37. Rizza, Robert J. *An Object-Oriented Military Simulation Baseline for Parallel Simulation Research.* MS thesis, AFIT/GCS/ENG/90D-12, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1990. AD-A231030.

38. Roberts, Stephen D. and Joe Heim. "A Perspective on Object-Oriented Simulation." *Proceedings of the 1988 Winter Simulation Conference*, edited by Michael A. Abrams and others. 277–81. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, December 1988.

39. Rumbaugh, James and others. *Object-Oriented Modeling and Design.* Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1991.

40. Sadayappan, P. and others. "Cluster Partitioning Approaches to Mapping Parallel Programs onto a Hypercube," *Parallel Computing*, *13*(1):1–16 (January 1990).

41. Sartor, JoAnn M. *Optimal Iterative Task Scheduling for Parallel Simulations.* MS thesis, AFIT/GCS/ENG/91M-03, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, March 1991. AD-A238631.

42. Schuppe, Thomas F. "Modeling and Simulation: A Department of Defense Critical Technology." *Proceedings of the 1991 Winter Simulation Conference*, edited by Barry A. Nelson and others. 519–25. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, December 1991.

43. Sha, Lui and Shirish S. Sathaye. "A Systematic Approach to Designing Distributed Real-Time Systems," *IEEE Computer*, *26*(9):68–78 (September 1993).

44. Shirazi, Behrooz and others. "Analysis and Evaluation of Heuristic Methods for Static Task Scheduling," *Journal of Parallel and Distributed Computing*, *10*(3):222–32 (November 1990).

45. Shirazi, Behrooz and A. R. Hurson. "Special Issue on Scheduling and Load Balancing; Guest Editors' Introduction," *Journal of Parallel and Distributed Computing*, *16*(4):271–4 (December 1992).

46. Soderholm, Steven R. *A Hybrid Approach to Battlefield Parallel Discrete Event Simulation.* MS thesis, AFIT/GCS/ENG/91D-23, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1991. AD-A24375.

47. Som, Tapas K. and others. "On Analyzing Events to Estimate the Possible Speedup of Parallel Discrete Event Simulation." *Proceedings of the 1989 Winter Simulation Conference*, edited by Edward A. MacNair and others. 729–37. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, December 1989.

48. Stanat, Donald F. and David F. McAllister. *Discrete Mathematics in Computer Science.* Englewood Cliffs, New Jersey 07632: Prentice-Hall, 1977.

49. Steinman, Jeff S. "Discrete-Event Simulation and the Event Horizon." *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, edited by D. K. Arvind and others. 39–49. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, July 1994.

50. Steinman, Jeff S. and Frederick Wieland. "Parallel Proximity Detection and the Distribution List Algorithm." *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, edited by D. K. Arvind and others. 3–11. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, July 1994.

51. Trachsel, Walter Gordon. *Object Interaction in a Parallel Object-Oriented Discrete-Event Simulation.* MS thesis, AFIT/GCS/ENG/93D-22, School of Engineering, Air

Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1993. AD-A274084.

52. Van Horn, Prescott John. *Development of a Protocol Usage Guideline for Conservative Parallel Simulations.* MS thesis, AFIT/GCS/ENG/92D-19, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson Air Force Base OH, December 1992. AD-A258851.

53. Wieland, Frederick and others. "An Empirical Study of Data Partitioning and Replication in Parallel Simulation." *Proceedings of the Fifth Distributed Memory Computing Conference*, edited by David W. Walker and Quentin F. Stout. 916–21. 10662 Los Vaqueros Circle, PO Box 3014, Los Alamitos CA 90720-1264: IEEE Computer Society Press, April 1990.

54. Woodside, C. Murray. "Fast Allocation of Processes in Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, *4*(2):164–74 (February 1993).

Vita

Captain James B. Hiller was born September 11, 1964, in Mount Kisco, New York and grew up in Tyrone, Pennsylvania. In May 1985, he graduated with a Bachelor of Science degree in Computer Science from Worcester Polytechnic Institute and an Air Force commission as a Second Lieutenant.

Captain Hiller's first assignment was to the Space and Warning Systems Center at Peterson Air Force Base as a Missile Warning Software Analyst. He was responsible for communication software maintenance for the Mission-Essential Backup (MEBU) system supporting the North American Aerospace Defense Command (NORAD). In 1988, Captain Hiller became the Network Software Security Officer responsible for system security engineering and management for all space, missile warning, communication systems maintained by the SWSC.

In 1991, Captain Hiller was assigned to the Air Force Cryptologic Support Center, Air Force Intelligence Command, at Kelly Air Force Base. He served as C4 Systems Security Policy and Doctrine Officer, responsible for development of Air Force system security policy and technical support to major commands. Captain Hiller continued in this assignment until his arrival at AFIT in May 1993.

While at AFIT, Captain Hiller was granted professional status as a Certified Information Systems Security Professional (CISSP) by the International Information Systems Security Certification Consortium.

Permanent address: 3 Laurel Drive
PO Box 95
Tyrone, PA 16686

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1994	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE Analytic Performance Models for Parallel Discrete Event Battlefield Simulation with Conservative Synchronization	5. FUNDING NUMBERS
6. AUTHOR(S) James B. Hiller, Captain, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/94D-08
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Capt Rick Painter 2241 Avionics Circle, Suite 16 WL/AAWA-1 BLD 620 Wright-Patterson AFB, OH 45433-7765 (513)-255-4429	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This study investigated the development and use of analytic models for performance analysis of parallel discrete event battlefield simulation using conservative synchronization. A simulation architecture with layered application, simulation, and host machine services provided the model development basis. Simulation entities were modeled with set-theoretic definitions. Deterministic performance models using these definitions were developed for event prediction, scheduling, and execution in sequential battlefield simulation. The sequential model was expanded to include relative bounds for overhead factors introduced when the simulation is spatially decomposed for a parallel distributed memory machine. Comparison of sequential and parallel models instantiated for a simulation with uniform workload showed a potential for unbounded processor blocking. A synchronization algorithm modification to limit per-iteration blocking is shown theoretically to decrease finishing time. Modification results were demonstrated on a hypercube architecture. Demonstration showed that a sequential simulation requiring 60 seconds to run was limited to a best time of 30 seconds on four processors without algorithm modification. The time was improved to 17 seconds using the modification. A number of basic timing measurements also showed that event list operations on a sequential structure take significantly longer than interactive event prediction algorithms using simulation entities maintained in similar structures.

14. SUBJECT TERMS Parallel, Simulation, Performance, Modeling	15. NUMBER OF PAGES 153
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL