Simulation techniques in an artificial society model

Barry Glenn Lawson
College of William & Mary - Arts & Sciences

SIMULATION TECHNIQUES IN AN ARTIFICIAL SOCIETY MODEL

A Dissertation

Presented to

The Faculty of the Department of Computer Science

The College of William & Mary in Virginia

In Partial Fulfillment

Of the Requirements for the Degree of

Doctor of Philosophy

by

Barry Glenn Lawson

2002

APPROVAL SHEET

This dissertation is submitted in partial fulfillment of

the requirements for the degree of

Doctor of Philosophy

[illegible]

Barry G. Lawson

Approved, June 2002

Stephen K. Park (R. H. Prosl)

Stephen K. Park
Thesis Advisor

Richard H. Prosl

Richard H. Prosl
Committee Chair

[illegible]

Gianfranco Ciardo

Weizhen Mao

Weizhen Mao

[illegible]

Evgenia Smirni

Lawrence Leemis

Lawrence Leemis
Department of Mathematics

To my family, for their love, support, and patience...

Table of Contents

Acknowledgments **ix**

List of Tables **x**

List of Figures **xvi**

Abstract **xvii**

1 Introduction **2**

1.1 Background 3

1.2 Motivation 5

1.3 Overview 7

2 Model Description **8**

2.1 Conceptual Model 9

2.2 Specification Model 11

2.2.1 Model Parameters 11

2.2.2 The Landscape 12

2.2.3 The Agents 14

2.2.4 Behavioral Rules 17

2.2.5 Time Evolution 20

2.2.5.1 Synchronous Time Evolution 21

2.2.5.2 Asynchronous Time Evolution 22

2.3 Computational Model 28

2.3.1 Implementing the Landscape 29

2.3.2 Implementing the Agents 30

2.3.3 Implementing the List of Events 31

2.3.4 Implementing the Simulation 32

3 Artificial Society Model Output **34**

3.1 Initial Conditions 35

3.2 Landscapes 36

3.3 Experimental Results 38

3.3.1 Basic Output for Two-Peak Landscape 39

3.3.2 Output for Two-Peak Landscape with Large T 42

3.3.3 Output for Two-Peak Landscape with Large $A(0)$ 45

3.3.4 Output for Large Dimension Two-Peak Landscape 45

3.3.5 Output using Other Landscapes 47

3.3.6 Output using Alternate Fields of View and Cell Shape 50

3.3.7 Output using Non-Periodic Landscape Boundary Conditions 52

3.3.8 Output with Movement and Reproduction Uncoupled 54

3.3.9 Output Summary 58

4 **The Event List** **59**

4.1 Background 60

4.2 Event List Implementations 61

4.2.1 Sorted Singly-Linked List 62

4.2.2 Henriksen's Algorithm 63

4.2.3 Splay Tree 65

4.2.4 Calendar Queue 69

4.2.5 Spatially Based Multilists 72

4.3 Event List Performance 75

4.4 Event List Summary 79

5 **Job Scheduling in Parallel Systems** **80**

5.1 Background 81

5.2 Scheduling Workload Analysis 83

5.2.1 The Arrival Process 84

5.2.2 The Service Process 85

5.2.2.1 Classification Using Accurate Service Estimates 87

5.2.2.2 Classification Using Inaccurate Service Estimates 89

5.3 Scheduling Policies 91

5.3.1 Single-Queue Backfilling 91

5.3.2 Multiple-Queue Backfilling Using Accurate Estimates 93

5.3.3 Multiple-Queue Backfilling Using Inaccurate Estimates 96

5.3.4 Backfilling with Job Priorities and Reservations 98

5.3.4.1 Single-Queue Backfilling with Priorities 99

5.3.4.2 Multiple-Queue Backfilling with Priorities 100

5.3.4.3 Backfilling with Reservations 100

5.3.5 Scheduling Policies Summary 101

5.4 Performance Analysis 101

5.4.1 Backfilling Policy Comparison Using Accurate Estimates 103

5.4.2 Backfilling Policy Comparison Using Inaccurate Estimates 105

5.4.3 Backfilling Policy Comparison Under Heavy Load 109

5.4.4 Backfilling Policy Comparison Under Job Priorities 111

5.4.5 Backfilling Policy Comparison Under Reservations 117

5.4.6 Policy Comparison Summary 119

6 Summary and Future Work **120**

A Random Variable Models **123**

A.1 Uniform 123

A.2 Exponential 124

A.3 Equilikely 125

A.4 Bernoulli 126

A.5 Random Variate Generator Code 127

B Related Simulation Models **131**

B.1 Soil Erosion Model 131

B.2 Prisoners' Dilemma Model 138

Bibliography **142**

ACKNOWLEDGMENTS

First, I would like to acknowledge my thesis advisor, the late Steve Park, for the tutelage, the conversation, and especially his "philosophy ... for free". May I influence and inspire a generation of students in the same manner as he.

Special thanks goes to my committee chair, Dick Prosl, for always looking out for me. I also want to thank the following faculty members for their assistance and understanding during this process: Gianfranco Ciardo, Larry Leemis, Weizhen Mao, Evgenia Smirni, and Andreas Stathopoulos. Thanks to Vanessa Godwin and the myriad of assistants for their great support. Thanks also to Clinch Valley College professors Emmet Low and George Culbertson for their influence, and thanks to fellow CVC alum Lisa Stanley for her friendship.

Many friends I gained while at W&M deserve special recognition for providing interesting diversions, whether athletic, inspirational, musical, or philosophical: Jack Cowardin, Ernest and Gaëlle Frazer, Jessen and Beth Havill, Scott and Molly Hayes, Joel and Heidi Hollingsworth, Mark and Dawn Idema, Wenlei Mao, Jim McCombs, Rance Necaise, Gregg and Kia Rippel, Ken Sollars, Beverly Thompson, Ed Tober, Khurshid Usmani, Gregg Wheeler, the Fredericksburg Baseball Club, the Friday basketball crew, and Walnut Hills Baptist Church. I want to extend a special thank you to Missy Rau for being a great friend (and for proofreading).

I would like to express deepest appreciation to my parents, Leon and Peggy Lawson, for all they have done for me. Thanks to the rest of my family for their love and support. Most importantly, thanks to the Lord for the gift of life.

List of Tables

3.1 Proportion of replications that exhibit oscillatory behavior using asynchronous time evolution with agent movement and reproduction uncoupled, and gestation period $\eta = 0.0$. All landscapes have common resource mean of 128. . . 55

4.1 Time complexities for sorted linked list 63

4.2 Time complexities for Henriksen's algorithm 65

4.3 Time complexities for Sleator-Tarjan splay tree 67

4.4 Time complexities for Jones-modified splay tree 68

4.5 Spatially based multilist sublist assignments with $S = 3$ 74

4.6 Spatially based multilist sublist assignments with $S = 4$ 74

5.1 Summary statistics of the four selected workloads. All times are reported in seconds. 86

5.2 Proportions of (possibly) crashed jobs for three parallel workload traces . . 98

5.3 Number of missed reservations for single- and multiple-queue backfilling with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations . 118

B.1 Payoff p to cell (x_1, y_1) encountering cell (x_2, y_2) 138

List of Figures

2.1 $X \times Y$ landscape grid 12

2.2 A 50×50 landscape using $\gamma(x, y)$ as defined in Equation 2.2 14

2.3 The FOV for an agent at (x, y) if $\phi = 3$ 15

2.4 Landscape from Figure 2.2 with 400 initial agents placed at random 17

2.5 The one-cell von Neumann neighborhood for an agent at (x, y) 19

2.6 Typical agent wealth synchronous time history 22

2.7 Typical agent wealth asynchronous time history 28

3.1 The landscapes of varying resource capacity distributions used in our experiments 37

3.2 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 50×50 two-peak landscape with 0.16 of the total cells initially populated at random 40

3.3 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 100×100 two-peak landscape with 0.16 of the total cells initially populated at random 41

3.4 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 25\,000$ on a 50×50 two-peak landscape with 0.16 of the total cells initially populated at random 43

3.5 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 25\,000$ on a 100×100 two-peak landscape with 0.16 of the total cells initially populated at random 44

3.6 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 50×50 two-peak landscape with initial population of 2000 agents placed at random 46

3.7 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on 500×500 and 1000×1000 two-peak landscapes with 0.16 of the total cells initially populated at random 47

3.8 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions with 0.16 of the total cells initially populated at random 48

3.9 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ and initial seed 12345 on 100×100 digital image landscapes with 0.16 of the total cells initially populated at random 49

3.10 Alternate Fields of View (FOV) for an agent at (x, y) if $\phi = 3$ 50

3.11 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity using alternate agent fields of view (FOV) for $T = 2500$ and initial seed 12345 on 100×100 two-peak landscape with 0.16 of the total cells initially populated at random 52

3.12 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity with non-periodic boundary conditions for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions and 0.16 of the total cells initially populated at random 53

3.13 Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions with 0.16 of the total cells initially populated at random. For asynchronous time evolution, agent movement and mating are uncoupled, with gestation period $\eta = 1.0$. 57

4.1 A sample singly-linked list event list 62

4.2 A sample Henriksen's event list 64

4.3 A sample splay tree event list 66

4.4 Rotation for node r with parent q but no grandparent 67

4.5 RR rotation for splay operation 68

4.6 RL rotation for splay operation 69

4.7 A sample calendar queue event list 70

4.8 A sample multilist event list schematic using Henriksen's sublists 72

4.9 Execution times using sorted linked list and Henriksen's event list implementations with $p = 0.8$ of the total cells occupied for increasing landscape sizes 76

4.10 Execution times using Henriksen's, splay tree, and calendar queue event list implementations with a proportion $p = 0.8$ of the total cells occupied for increasing landscape sizes 77

4.11 Execution times using spatially-based multilist event list implementations with a proportion $p = 0.8$ of the total cells occupied for increasing landscape sizes 78

5.1 Total number of arriving jobs per week as a function of time (weeks) 85

5.2 Actual service time characteristics of the four workloads using four-part classification 88

5.3 Service time overestimates as a function of time 90

5.4 An example 32-processor system in which multiple-queue backfilling permits the four initial partition boundaries to adapt as workload conditions change 95

5.5 Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces. Using actual service times as accurate estimates, we compare multiple-queue backfilling (with four-part classification) to single-queue backfilling, both without speculative execution. 104

5.6 Slowdown ratio $\mathcal{R}$ per 1000-job submissions as a function of time for each of the four traces. Using actual service times as accurate estimates, we compare multiple-queue backfilling (with four-part classification) to single-queue backfilling, both without speculative execution. 106

5.7 Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces using inaccurate user estimates. Using user-provided inaccurate estimates, we compare multiple-queue backfilling (with three-part classification) to single-queue backfilling, both employing speculative execution. 107

5.8 Slowdown ratio $\mathcal{R}$ per 1000-job submissions as a function of time for each of the four traces. Using user-provided inaccurate estimates, we compare multiple-queue backfilling (with three-part classification) to single-queue backfilling, both employing speculative execution. 108

5.9 Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with increasing system load. All slowdown ratios are computed relative to single-queue backfilling under the same load. 110

5.10 Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces where $p = 0.75$ of the total jobs have high priority 112

5.11 Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces where $p = 0.75$ of the total jobs have high priority 114

5.12 Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces where $p = 0.95$ of the total jobs have high priority 115

5.13 Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces where $p = 0.95$ of the total jobs have high priority 116

5.14 Overall aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations 117

5.15 Distribution tails of the delays experienced by jobs requesting reservations with a proportion of 0.25 of the total jobs requesting reservations 118

A.1 Probability density function of a $\mathit{Uniform}(a, b)$ random variable 124

A.2 Probability density function of an $\mathit{Exponential}(\mu)$ random variable 125

A.3 Probability density function of an $\mathit{Equilikely}(a, b)$ random variable 126

A.4 Probability density function of a $\mathit{Bernoulli}(p)$ random variable with $p = 0.6$ 126

B.1 Effect of stabilize rule on center cell 133

B.2 Effect of develop rule on center cell 133

B.3 Example 50×50 landscape to be developed 136

B.4 Synchronous application of the develop rule to Figure B.3(b) 137

B.5 Asynchronous application of the develop rule to Figure B.3(b) 137

B.6 Synchronous application of the Prisoners' Dilemma 140

B.7 Asynchronous application of the Prisoners' Dilemma, $t = 30$ 141

ABSTRACT

"Artificial society" refers to a generic class of agent-based simulation models used to discover global social structures and collective behavior produced by simple local rules and interaction mechanisms. Artificial society models are applicable in a variety of disciplines, including the modeling of chemical and biological processes, natural phenomena, and complex adaptive systems. We focus on the underlying simulation techniques used in artificial society discrete-event simulation models, including model time evolution and computational performance.

Although for some applications synchronous time evolution is the correct modeling approach, many other applications are better represented using asynchronous time evolution. We claim that asynchronous time evolution can eliminate potential simulation artifacts produced using synchronous time evolution. Using an adaptation of a popular artificial society model, we show that very different output can result based solely on the choice of asynchronous or synchronous time evolution. Based on the event list implementation chosen, the use of discrete-event simulation to incorporate asynchronous time evolution can incur a substantial loss in computational performance. Accordingly, we evaluate select event list implementations within the artificial society simulation model and demonstrate that acceptable performance can be achieved.

In addition to the artificial society model, we show that transforming from a synchronous to an asynchronous system proves beneficial for scheduling resources in a parallel system. We focus on non-FCFS job scheduling policies that permit jobs to backfill, i.e., to move ahead in the queue, given that they do not delay certain previously submitted jobs. Instead of using a single queue of jobs, we propose a simple yet effective backfilling scheduling policy that effectively separates short from long jobs by incorporating multiple queues. By monitoring system performance, our policy adapts its configuration parameters in response to severe changes in the job arrival pattern and/or resource demands. Detailed performance comparisons via simulation using actual parallel workload traces indicate that our proposed policy consistently outperforms traditional backfilling in a variety of contexts.

SIMULATION TECHNIQUES IN AN ARTIFICIAL SOCIETY MODEL

Chapter 1

Introduction

An *artificial society* is an agent-based model used to study emergent processes as they evolve on a landscape. Such processes may include population dynamics, group formation, environmental and economic impacts, propagation of disease, cultural influences, and combat. The artificial society is an *in silico laboratory* [28] in which societies are "grown" in the computer. The goal is to realize complex collective (i.e., macroscopic) social behaviors of interest that result from the behavior of heterogeneous individuals controlled by relatively simple rules defined on a local (i.e., microscopic) level.

Generally, artificial society simulation models consist of three components: a landscape, agents, and a set of behavioral rules. The landscape is typically a two-dimensional grid of cells containing a heterogeneous distribution of one or more resources of interest to the agents. The agents are rational actors that move about on the landscape, interacting with the landscape and other agents. Together, the agents comprise the population to be studied, e.g., people or insects. The time-evolved progression of the agents on the landscape is controlled by behavioral rules that determine how agents interact with the landscape and with one another.

The complexity that results from the combination of these components within a simulation model is easily overlooked. Indeed, serious consideration must be given to the underlying simulation details because in many applications artifacts in the output can result. In this work, for applications involving natural asynchronous behavior we promote asynchronous time evolution, in which different event types occur at random at their own characteristic rates. We investigate in detail the impact of time evolution in a popular, representative artificial society model, and examine the corresponding simulation issues that arise.

1.1 Background

The idea of simulating artificial societies has its foundations in cellular automata based work. Cellular automata (CA) were introduced in the late 1940s by John von Neumann [107] and Stanislaw Ulam. In the context of physical systems, Arthur Burks [14] subsequently completed and extended the work started by von Neumann. Cellular automata became even more popular with the advent of John Conway's "Game of Life" [34] and gave rise to the use of cellular automata in games. Another example of the use of cellular automata, and perhaps the first to apply CA in a societal application, was Thomas Schelling's model to study segregation [96].

More recent works by Gutowitz [43], Toffoli and Margolus [105], and Wolfram [110, 111] considered theoretical aspects and scientific applications of cellular automata, including mathematical analysis, computation theory, and various physical, chemical, and biological applications. More specifically, cellular automata based models have been used to study percolation [41], earthquakes and forest fires [5, 6, 42], soil erosion, and fluid dynamics [24, 105]. Cellular automata have also been used in the study of complex adaptive systems [49], cooperative behavior (e.g., the Prisoners' Dilemma model [2, 27, 45, 80]), and population dynamics [33].

The term *artificial society*, originating in work by Builder and Bankes at the RAND Corporation in the early 1990s [11], refers to an agent-based model used to study emergent processes. Interest in such models increased as a result of the 2050 Project, in which research centers such as the Brookings Institution and the Santa Fe Institute were commissioned to study and predict the evolution of society under the threat of ever-depleting natural resources. In recent years, agent-based models have achieved growing popularity in applications from many different disciplines. Agent-based models have been used in applications from the social sciences (e.g., [28, 36, 37, 38, 39, 53, 79]), in military applications (e.g., [66, 89]), and in applications in the "hard" sciences, including biology, chemistry, ecology, engineering, geography, and marine biology (e.g., [25, 26, 57, 65, 72]). Moreover, work based on the same agent-based model and conducted parallel to the work presented herein has been the topic of several recent graduate-level theses at William and Mary [44, 75, 76].

Several languages and software packages have been developed for implementing artificial society simulation models. For example, Mitchell Resnick at MIT marketed software to the general public [88]. Swarm [73], an object-oriented package developed at the Santa Fe Institute, is a popular means for implementing artificial society simulations (e.g., [72, 103]). The objective of the Swarm simulation system is to provide as a standard an easy-to-use language for agent-based simulations. Other modeling languages of varying levels of complexity include AgentSheets [87], Ascape [83], ARVA (developed at the University of Caen, France), MIMOSE (Micro- and Multilevel Modeling Software) [74], SDML (Strictly

Declarative Modeling Language) [77], and UMDBS (Universal Micro Data Base System) [95].

Artificial society models are popular because social processes and emergent behavior are often too complex to study by traditional methods. Historically, social scientists attempt to study social behavior by dividing processes into individual subprocesses and combining the individual observations. However, such approaches are difficult to address scientifically because the effects of the individual subprocesses and their complex interactions are not well defined [44]. Furthermore, attempts to use mathematical models fail to fully encapsulate the complexity available via simulation. Admittedly, the use of simulation models to study complex social behavior has its detractors [112], but the ubiquity of such models coupled with explicit requests for simulation approaches [19, 36, 45, 46, 68] give merit to this ever-growing field of study.

1.2 Motivation

The work herein began in an attempt to reproduce the results from the artificial society simulation model presented in [28]. Despite considerable experimentation, we were never able to reproduce all the results. These failures prompted us to evaluate and question the underlying simulation approaches used by the authors. We were further motivated by Wilson [109] who also questioned the scientific nature of the results. We determined that many of the modeling decisions used in [28] are unnatural, ambiguous, and produce artifacts in the results. Accordingly, we began to search for studies that address these problems.

Works in related journals and edited collections (e.g., [37, 38, 54]) generally focus on the design of various behavioral rules or the application of models to specific societal applications. The use of genetic algorithms and artificial intelligence is also a common theme [45, 60, 79, 108]. Little, if any, significance is placed on the details of the simulation model at the computational level. In fact, a scientific approach to simulation is noticeably absent in recent works [4, 27, 28, 53, 103]. The availability of custom modeling languages, such as Swarm, leads some modelers to dismiss simulation issues [103]. Such languages claim to simplify and standardize the implementation of agent-based models, but do not promote or facilitate an investigation of the underlying simulation techniques. Although the languages may have the capability to model asynchronous behavior (e.g., [12, 13]), we were unable to find any detailed investigations exploiting this capability.

Müller et al. [79] specifically request interdisciplinary coordination so that the "soft" sciences can benefit from the scientific approaches commonly used in the "hard" sciences. Most recent related works by authors in the "hard" sciences focus on efficient parallel implementations of agent-based models [20, 21, 22, 40, 104]. The use of asynchronous time evolution is noted in some recent studies [9, 20, 23], but no investigation of the effects of time evolution and the other questionable simulation issues from [28] is provided. Aside from the work presented herein and published elsewhere [62], such an investigation is absent from the literature. For this reason, we investigate in detail the effects of the time evolution of a representative artificial society model and the associated simulation issues that arise.

1.3 Overview

The remainder of this dissertation is organized as follows. In Chapter 2, a detailed description of the artificial society simulation model used throughout is presented. This model is based on the extensive artificial society model defined in [28]. We use a subset of the behavioral rules defined in [28], and modify these rules to be more realistic. In Chapter 3, we present results from our simulation model that exhibit very different behavior based solely on the choice of synchronous or asynchronous time evolution of the model. Through detailed investigations we show that different behavior can result for a variety of initial conditions and model parameters. Because the required implementation can result in very poor performance, in Chapter 4 we present appropriate data structures and associated algorithms for the implementation. Using our artificial society simulation model, we evaluate and compare the performance of each of these data structures, and show that acceptable computational performance can be achieved. In Chapter 5, we show that, like the artificial society model, transforming from a synchronous to an asynchronous system also improves job scheduling for resources in parallel systems. We describe in detail a new policy for job scheduling, and we provide results showing that our policy outperforms the standard policy in a variety of contexts. Finally, in Chapter 6 we provide concluding remarks and areas for future research.

Chapter 2

Model Description

We use the term "artificial society" to refer to an agent-based model used to study social processes as they evolve on a landscape. Among these processes are agent interaction with the landscape, population dynamics (including birth, death and reproduction), economic, cultural and disease interactions, and combat. The artificial society is in essence a "computational laboratory" in which the evolution of social processes can be studied. The underlying motivation is an attempt to discover the emergence of interesting *macro*-scale social structures and collective behavior by defining simple *micro*-scale rules and interaction mechanisms.

Because our goal is to promote an alternative approach for artificial society discrete-event simulation models, *not* to develop fundamentally new models, we base our work on an adaptation of the extensive artificial society model presented in [28]. To demonstrate that different results can be obtained using asynchronous versus synchronous time evolution in an artificial society model, we balance functionality with model complexity by including only a subset of the rules defined by Epstein and Axtell. In some instances, we modify these rules to be more realistic. Our model can be easily extended to include additional rules.

The organization of this chapter is based on the principle that models should be developed at three levels — conceptual, specification, and computational[1]. The development of a model at the conceptual level involves describing and understanding the model at a high level. The specification level of a model incorporates variables, equations, algorithms and logic to describe the model in more detail. The computational level of a model is characterized by the actual implementation of the model as a computer program. A comprehensive definition of our artificial society model follows.

2.1 Conceptual Model

Consider a discrete-event simulation model comprised of agents inhabiting a landscape. As time evolves, the agents move about the landscape interacting with the landscape and with one another. All *actions* and *interactions* between the landscape and the agents are controlled by a set of micro-scale rules. The model is characterized by a global clock, by the number and location of agents on the landscape, and by the spatial distribution of landscape resources.

The landscape in our artificial society model is the environment *in* which the agents operate and *with* which they interact. The landscape consists of a two-dimensional grid of cells containing a heterogeneous distribution of one or more distinct resources. These resources are gathered and consumed by the agents to survive. Each landscape cell may be occupied by at most one agent at any time. At each cell the level of each resource changes with time as it is gathered and consumed by the agents. Each resource at each cell has an associated regrowth rate and maximum capacity. As regrowth occurs, depleted landscape

[1]See Algorithm 1.1.1 of [82].

resources are replenished.

Agents gather and consume resources as they move about the landscape. At any time during the evolution of the model, each agent has a distinct location on the landscape, characterized by the cell the agent occupies, and a distinct field of view, measured in landscape cells. Each agent has perfect knowledge of other agents and of landscape resource levels within its field of view; an agent has no knowledge or memory of agents and landscape resource levels outside its field of view. In addition, an agent has a characteristic metabolic rate for each resource it gathers and consumes. Any resources gathered by an agent, but not consumed, are retained for future use and, in this way, an agent's "wealth" can grow without constraint. If an agent's wealth diminishes to zero, the agent dies from starvation.

As time evolves, three micro-scale behavioral rules control the actions and interactions between the landscape cells and the agents. Under the *regrowth* rule, a depleted resource at a given cell is replenished up to the maximum capacity of that cell. Under the *movement* rule, an agent moves to the unoccupied cell with maximum resource within its field of view, then gathers and consumes the resources at that cell. Under the *reproduction* rule, neighboring fertile agents of the opposite sex breed and, in this way, produce new agents.

The model is initialized by placing on the landscape a random distribution of agents, each with characteristic attributes and initial states. Consistent with the behavioral rules, as time evolves agents move about the landscape interacting with one another and with the landscape. Statistics are gathered during the model's lifetime to provide data for analyzing the resulting macro-scale behavior of the agents.

2.2 Specification Model

The following definitions provide the basis for developing the artificial society model at the specification level. Both the landscape and agents are defined by dynamic *states*, which change with time, and static *attributes*, fixed for the lifetime of the model. In the notation to follow, we adopt the convention of using uppercase letters to represent cardinality parameters, lowercase letters to represent dynamic states, and lowercase Greek symbols to represent static attributes.

2.2.1 Model Parameters

In anticipation of providing a detailed description of the artificial society model, we define the model attributes

T:	the maximum simulated time for the model;
X, Y:	the vertical and horizontal landscape dimensions respectively;
N:	the number of distinct landscape resources

and model variables

t:	time[2] $(0 \le t \le T)$;
$A(t)$:	the number of agents on the landscape at time t.

Following convention, time begins at $t = 0$ and ends when $t = T$. The landscape dimensions X and Y, fixed for the lifetime of the model, are quantified in landscape cells and are naturally discrete. The number of agents $A(t)$ is a function of time, and is also naturally discrete. The number of resources N is fixed for the lifetime of the model, and is naturally

[2]Time can be modeled as either synchronous (fixed-increment time steps) or asynchronous (random event times). We defer the discussion of synchronous and asynchronous time evolution until Section 2.2.5.

discrete. When necessary, we use subscript $a = 0, 1, \ldots, A(t) - 1$ to distinguish agents and subscript $n = 0, 1, \ldots, N - 1$ to distinguish resources.

2.2.2 The Landscape

An (x, y) integer-valued spatial coordinate system is used to identify cells in the landscape. The landscape is a two-dimensional grid with dimensions $X \times Y$. Define $\mathcal{X} = \{0, 1, \ldots, X - 1\}$ and $\mathcal{Y} = \{0, 1, \ldots, Y - 1\}$. Then each cell in the $X \times Y$ landscape is distinguished by its (x, y) position with $(x, y) \in \mathcal{X} \times \mathcal{Y}$. We rotate the coordinate system 90 degrees relative to a conventional (x, y) orientation so that increasing x indexes rows from top to bottom and increasing y indexes columns from left to right, as shown in Figure 2.1.

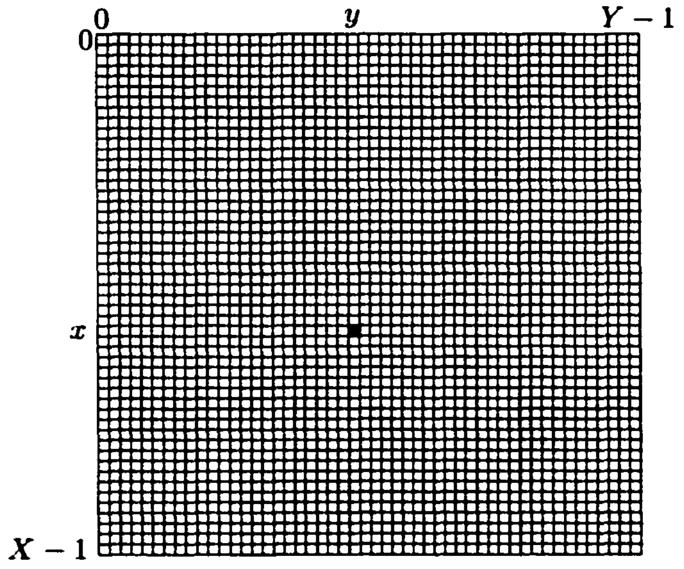


Figure 2.1: $X \times Y$ landscape grid

The landscape is subject to periodic boundary conditions. That is, the four boundaries wrap around from right to left and from bottom to top, creating a torus. For an $X \times Y$ landscape, any (x', y') cell that lies beyond the landscape boundaries, i.e., $(x', y') \notin \mathcal{X} \times \mathcal{Y}$, is equated to a corresponding cell $(x, y) \in \mathcal{X} \times \mathcal{Y}$ within the boundaries via the equation

$$(x, y) = (x' \mod X,\ y' \mod Y). \tag{2.1}$$

For example, if $X = Y = 50$ then cells $(-1, 20)$ and $(49, 20)$ are equivalent and cells $(52, 50)$ and $(2, 0)$ are equivalent.

Each (x, y) cell in the landscape is characterized by the attributes

$\gamma_n(x, y)$:	the capacity of resource n at cell (x, y);
$\rho_n(x, y)$:	the regrowth rate of resource n at cell (x, y)

and the state variables

$r_n(x, y, t)$:	the level of resource n at cell (x, y) at time t;
$d(x, y)$:	the most recent time of resource depletion at cell (x, y);
$o(x, y, t)$:	the occupancy status of cell (x, y) at time t.

The attributes $\gamma_n(x, y)$ and $\rho_n(x, y)$, and the state variable $r_n(x, y, t)$ are quantified in real-valued units of resource. The state variable $d(x, y)$ is a quantified in units of time. Because at most one agent can occupy any cell at time t, $o(x, y, t)$ is modeled naturally as Boolean.

In the remainder of this discussion, for simplicity we let $N = 1$ so that agents consider only one resource of interest on the landscape. This permits us to omit the n subscripts in our notation, with the understanding that $n = 0$ if the n subscript is omitted.

With this assumption, we define the real-valued resource capacity at an (x, y) landscape cell by the equation

$$\gamma(x, y) = f(x - X/4,\ y - Y/4) + f(x - 3X/4,\ y - 3Y/4) \tag{2.2}$$

ϕ:	the field of view (FOV) attribute;
μ:	the metabolic rate of resource consumption;
σ:	sex (male or female);
β:	the time of birth;
λ:	lifespan;
α:	the age when reproductive capability begins (puberty);
ω:	the age when reproductive capability ends ($\alpha < \omega$);
η:	the gestation period (for females)

and the state variables

$w(t)$:	the amount of resource wealth (holdings) at time t;
(x, y):	the landscape cell currently occupied by the agent;
$m(t)$:	pointer to the current mate (if any).

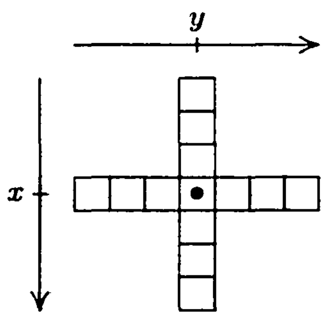


Figure 2.3: The FOV for an agent at (x, y) if $\phi = 3$

As illustrated in Figure 2.3, the field of view (FOV) attribute ϕ quantifies how far an agent can "see" in each of the four primary grid directions (north, south, east, and west) from the current (x, y) location. ϕ is naturally integer-valued and positive; typically $\phi \ll \min\{X, Y\}$. The metabolic rate μ is quantified in real-valued units of resource consumed per unit time. The sex of an agent σ is naturally modeled as Boolean. The remaining attributes are quantified in units of time. Death occurs naturally at time $\beta + \lambda$, but may occur prematurely as a result of starvation.

An agent's wealth $w(t)$ is quantified in real-valued units of resource, and can be arbitrarily large. The current location (x, y) of an agent is naturally represented as an integer-valued coordinate pair. If an agent successfully mates with another agent at time t, the state $m(t)$ points to the mate; otherwise, $m(t)$ is undefined.

Acceptance-rejection is used to generate the initial distribution of agents on the landscape[4]. Given an $X \times Y$ landscape, the (x, y) landscape cells to be occupied at time $t = 0$ by the $A(0)$ initial agents are determined according to Algorithm 2.1 to follow.

```
/* initially $o(x, y, t)$ = false for all $(x, y)$ at time $t = 0$ */
for ($a = 0$; $a < A(0)$; $a$++) {
      do {
         $x = Equilikely(0, X - 1)$;[5]
         $y = Equilikely(0, Y - 1)$;
      } while ( $o(x, y, 0)$ );
      $o(x, y, 0)$ = true;
}
```

Algorithm 2.1: Initially distribute agents

Figure 2.4 is a graphical depiction of a random initial distribution of $A(0) = 400$ agents on the $X \times Y = 50 \times 50$ landscape shown in Figure 2.2. Each agent is represented as an orange circle within a landscape cell.

[4]For a discussion of acceptance-rejection, refer to Section 7.6 of [82]. If the initial number of agents $A(0)$ is not small relative to the number of landscape cells, acceptance-rejection will be computationally inefficient.

[5]For a discussion of *Equilikely* random variates, refer to Section 6.4 of [82].

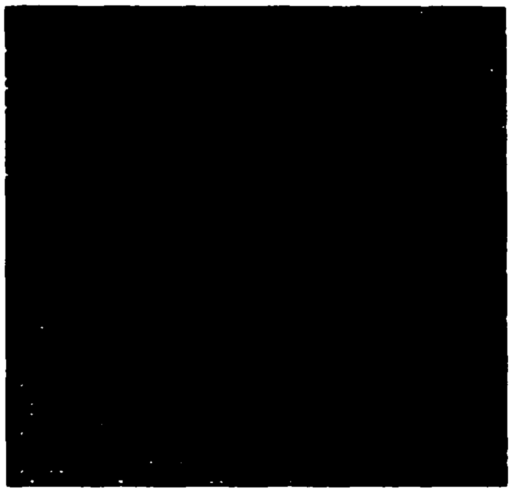


Figure 2.4: Landscape from Figure 2.2 with 400 initial agents placed at random

2.2.4 Behavioral Rules

Three types of behavioral rules control the actions and interactions of the landscape cells and agents [28]. *Agent-agent* rules govern both the actions of a single agent and the interaction of one agent with another. *Agent-environment* rules govern the interaction of an agent with one or more cells on the landscape. *Environment-environment* rules govern both the actions of a single landscape cell and the interaction of one landscape cell with neighboring cells. Each of the artificial society model rules defined below corresponds to exactly one of these three types.

We balance functionality with model complexity by including only a subset of the behavioral rules defined in [28]. In our artificial society model, we admit only the rules for resource regrowth, agent movement, and agent reproduction. Our model can be easily extended to include additional rules. Resource regrowth and agent movement remain as defined in [28]. We modify the agent reproduction rule to be more realistic, incorporating a gestation period[6]. Comprehensive definitions for each of the three rules follows[7].

Resource Regrowth Rule

The resource regrowth rule is an environment-environment rule defined as follows. At each unoccupied (x, y) cell, the resource is replenished at a rate of $\rho(x, y)$ resource units per unit time up to the maximum capacity $\gamma(x, y)$, so that for $t' > t$

$$r(x, y, t') = \min\Big\{\gamma(x, y),\ r(x, y, t) + (t' - t)\rho(x, y)\Big\}. \tag{2.4}$$

Agent Movement Rule

The agent movement rule is an agent-environment rule defined for each agent as follows.

- Look at all cells within the FOV defined by ϕ.
- Select the closest unoccupied (x, y) cell with maximum resource[8] and move to that cell[9].
- Immediately collect all resources currently at the cell.
- If $w(t) = 0$, where t is the current time of movement, the agent dies.

[6]If the gestation period $\eta = 0$, the behavior of agents under our reproduction rule is the same as if the gestation period was omitted from the rule.

[7]In the rule definitions to follow, we denote successive time steps as t and t' respectively where $t < t'$. We discuss time evolution in more detail in Section 2.2.5.

[8]In the unlikely case that two or more cells containing this local maximum are equidistant from the agent, the tie is broken stochastically.

[9]If $N > 1$, agents have a resource preference. Then, consistent with [28], the maximum resource at time t is given by

$$\max_{\forall(x,y)\in \mathrm{FOV}} \Big\{g(w_0(t) + r_0(x, y, t),\ w_1(t) + r_1(x, y, t),\ \ldots,\ w_{N-1}(t) + r_{N-1}(x, y, t))\Big\} \tag{2.5}$$

where

$$g(l_0, l_1, \ldots, l_{N-1}) = (l_0)^{\mu_0/M}(l_1)^{\mu_1/M} \cdots (l_{N-1})^{\mu_{N-1}/M}$$

with

$$M = \sum_{n=0}^{N-1} \mu_n.$$

Agent Reproduction Rule

The agent reproduction rule is an agent-agent rule. For reproduction occurring at time t, we define an agent to be fertile if all the following are true.

- The agent is of childbearing age, i.e., $\beta + \alpha \leq t < \beta + \omega$.
- The agent is not currently reproducing with any other agent, i.e., $m(t)$ is undefined.
- The agent will not die naturally by the end of the gestation period, i.e., $t + \eta < \beta + \lambda$.
- At the end of the gestation period, the agent will have amassed greater wealth than its initial wealth, i.e., $w(\beta) < w(t + \eta)$[10].

We then define the "responsible mate" mating algorithm for a fertile agent attempting to reproduce as follows.

- Select one of the four nearest neighbor agents (if any) of the opposite sex from the agent's one-cell von Neumann neighborhood, shown in Figure 2.5.

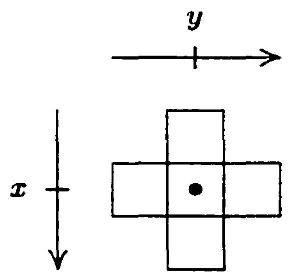


Figure 2.5: The one-cell von Neumann neighborhood for an agent at (x, y)

- If the selected neighbor agent is fertile and if there is an unoccupied cell (for the child) within the one-cell von Neumann neighborhood of either agent, the neighbor agent is a *candidate* for mating.

[10]The amount of wealth at the end of the gestation period is

$$w(t + \eta) = w(t) + \eta(\rho(x, y) - \mu). \tag{2.6}$$

- Repeat until the candidate with maximum wealth is found or the nearest neighbor search is exhausted with no available candidate.

If a mate is found, i.e., there is at least one candidate agent, the female agent of the pair becomes pregnant. Throughout the gestation period, neither the male nor the female agent can move or attempt further reproduction.

Consider a new agent $a = 3$ conceived at time t by parent agents $a = 1$ and $a = 2$ and born at time $t + \eta$. Each parent agent endows half of its initial wealth to the new agent so that, at time $t + \eta$, the wealth of agent $a = 1$ decreases by $w_1(\beta_1)/2$ resource units and the wealth of agent $a = 2$ decreases by $w_2(\beta_2)/2$ resource units. The initial wealth of the new agent at the time of birth $\beta_3 = t + \eta$ is the sum of the endowments from each parent agent according to

$$w_3(\beta_3) = \frac{w_1(\beta_1)}{2} + \frac{w_2(\beta_2)}{2}. \tag{2.7}$$

The sex of the new agent is chosen via a fair coin flip. The remaining attributes for the newborn agent are inherited directly from one of the parent agents chosen via a fair coin flip. At the time of birth, the unoccupied cell with maximum resource from the union of the one-cell von Neumann neighborhoods of both parents is selected for the child to occupy. If no such cell is available the child dies at the time of birth[11].

2.2.5 Time Evolution

Too often given little consideration in many artificial society discrete-event simulation models, time evolution is an important part of our research. We agree synchronous (fixed-

[11]If $\eta > 0$, during the gestation period other agents may occupy *all* cells adjacent to the parents. Consequently, at the time of birth a cell may not be available for the child to occupy.

increment) time evolution may be appropriate in certain models if, for example, movement is seasonally motivated. However, many other models are more accurately modeled using asynchronous evolution (random event times). In anticipation of presenting in Chapter 3 results obtained using both types of time evolution, a comprehensive definition of each type follows.

2.2.5.1 Synchronous Time Evolution

Synchronous time evolution involves fixed-increment time steps, with time conventionally denoted $t = 0, 1, 2, \ldots, T$. All events of interest must occur precisely at these time steps. For instance, in the Epstein and Axtell model all agents attempt to move simultaneously, once each time step. All agents attempting to reproduce must also do so once each time step. For serial program execution, this kind of parallel activity is not possible and so it is necessary to randomize the order in which agents act at each time step. Without this randomization, the agents will always act in the same order introducing a bias that can produce artifacts [28].

Given the model attribute T, which defines the number of simulated time steps, the artificial society model evolves synchronously in time according to Algorithm 2.2 to follow. Note the ambiguity in the order of events inside the while loop. For example, the modeler must decide whether the landscape should be updated *before* or *after* the agents move.

Figure 2.6 depicts a typical agent wealth time history for a synchronous time simulation model. Note the agent's wealth changes only at *fixed-increment* event occurrences according to

$$w(t) = \max\{0,\ w(t-1) + r(x,y,t) - \mu\}. \tag{2.8}$$

```
initialize the landscape;
initialize the agents;
$t = 0$;
while ($t \leq T$) {
   move and reproduce the agents;
   update the landscape;
   update the agent list;
   randomize (shuffle) the agent list;
   $t$++;
}
```

Algorithm 2.2: Synchronous time evolution

If $w(t) = 0$, the agent dies at time t.

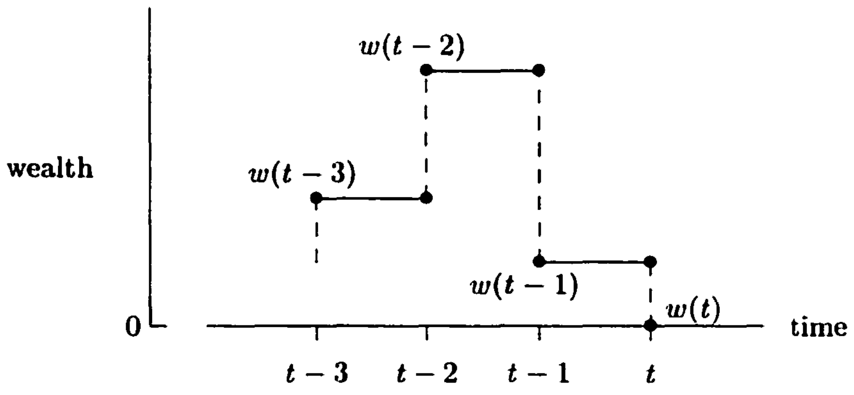


Figure 2.6: Typical agent wealth synchronous time history

2.2.5.2 Asynchronous Time Evolution

Asynchronous time evolution permits each type of event to occur at random at its own characteristic rate. In this way, individual agent movement events and reproduction events occur at distinct times. That is, whereas the order of agent actions must be specifically randomized at each time step in the synchronous case, randomization is inherent in the asynchronous case. In addition, in the asynchronous case there is no ambiguity in the order of events.

Next-Event Simulation Model

To facilitate asynchronous time evolution, we construct our artificial society model using a *next-event* simulation approach. For a next-event simulation model, we must clearly define the simulation clock, the system *state*, the event types, and a set of algorithms for each event type that define the state changes that occur when each type of event occurs [82].

The simulation clock is the real-valued global time parameter t. At any time $0 \leq t \leq T$, the following set of variables defines the system state.[12].

- $A(t)$;
- $o(x, y, t)$ and $r(x, y, t)$ and $d(x, y)$ for all $(x, y) \in \mathcal{X} \times \mathcal{Y}$;
- $(x, y)_a$ and $w_a(t)$ and $m_a(t)$ for all $a = 0, 1, \ldots, A(t) - 1$.

Accordingly, there are four types of events that can change the state of the system.

- Movement and corresponding resource consumption by agent a at time t will change $(x, y)_a$ and $w_a(t)$, $o(x, y, t)$ and $r(x, y, t)$ for the departed and newly occupied landscape cells, and $d(x, y)$ for the departed landscape cell.
- The death of agent a at time t will change $A(t)$ in addition to both $o(x, y, t)$ and $d(x, y)$ for $(x, y)_a$.
- A successful mating by agent a with another agent a' at time t will change $m_a(t)$ and $m_{a'}(t)$.

[12]For convenience, the state description includes some variables that could be determined by summing other variables. For instance, $A(t)$ can be determined by summing $o(x, y, t)$ for all $(x, y) \in \mathcal{X} \times \mathcal{Y}$. Accordingly, the state description is not minimal.

- The birth of a new agent a'' from parent agents a and a' at time t will change $w_a(t)$, $w_{a'}(t)$, $m_a(t)$, and $m_{a'}(t)$. If there is an unoccupied landscape cell for the new agent to occupy, the birth will change $A(t)$, create $(x, y)_{a''}$ and $w_{a''}(t)$, and change $o(x, y, t)$ and $r(x, y, t)$ for $(x, y)_{a''}$.

Because there is a gestation period, the state of the system can change when agents mate and when a new agent is born. For this reason, the reproduction rule, as defined in Section 2.2.4, is modeled by two distinct types of events. A mating event type initiates the gestation period if the female agent becomes pregnant. Since a mating event is an *attempt* to reproduce, the mating event does not guarantee pregnancy. A birth event type signals the end of the gestation period when the new agent is born (if possible). Both of these event types can change the state of the system as described above.

Asynchronous Inter-Event Times

For each event type, inter-event times are assumed to be *iid Exponential* $(1/\nu)$ where ν is the rate of occurrence of that event type. Equivalently, each event type is modeled as a stationary Poisson process with rate ν[13]. The mating event type can either be coupled with the movement event type or modeled as a separate stationary Poisson process. If the two event types are coupled, a mating event occurs for an agent each time a movement event occurs for that agent.

[13]It is common to assume that a stochastic process which occurs "at random" is a stationary Poisson process.

Asynchronous Resource Regrowth

Resource regrowth is not considered an event type, but rather an inherent part of the other event types. Consistent with [28], at $t = 0$ the resource level at each (x, y) cell is the same as the resource capacity at that cell. As time evolves, cell resources are depleted only when an agent occupies a cell. While a cell remains occupied, the resources continue to grow, but are continuously gathered by the occupying agent; at the moment an agent departs, the cell has no resources. As time evolves further, the resources can then regrow up to the resource capacity at that cell, provided the cell does not become occupied again. Because the onset of regrowth is a direct result of the interaction of an agent with a landscape cell, we include regrowth as part of the movement, mating, and birth event types.

For each (x, y) landscape cell, as time evolves $d(x, y)$ is defined by the last time an agent departed that cell. At time $t > d(x, y)$, we compute the current level of resource at cell (x, y) according to the equation

$$r(x, y, t) = \min\Big\{\gamma(x, y),\ \rho(x, y)(t - d(x, y))\Big\}. \tag{2.9}$$

Asynchronous Agent Movement

Consistent with the previous discussion of asynchronous resource regrowth, we now redefine the collection of resources by an agent under the movement rule. When an agent moves to an unoccupied cell, the agent immediately collects all resources currently at the cell. While occupying the cell, the agent continues to collect the resources as they regrow and consumes them sufficient to satisfy its metabolic rate. The wealth of the agent throughout this process of resource collection and consumption is computed as

$$w(t') = \max\left\{0, w(t) + r(x, y, t') - (t' - t)\mu\right\} \tag{2.10}$$

where t is the current time of movement and $t' > t$ is the next time of activity, either movement or reproduction, for the agent[14].

Asynchronous Time Evolution Algorithm

Given T, the artificial society model evolves asynchronously in time according to Algorithm 2.3 to follow. The algorithm presumes the existence of a function **dequeueEvent()** that returns the next (most imminent) event in simulated time. Within the algorithm, note each of the four types of events previously defined.

[14]Note that $r(x, y, t')$ accounts for both the initial resources collected at time t and the resources collected as they regrow during (t, t').

```
initialize the landscape;
initialize the agents;
$e$ = dequeueEvent();
while ($e$.time $\leq T$ and $A(t) > 0$) {
   switch ($e$.type) {
      case movement:
         vacate the current cell;
         select and occupy the $(x, y)$ cell within the FOV with maximum resource;
         update the wealth of the agent;
         schedule the next movement event for the agent;
      case death:
         $A(t)--$;
         vacate the current cell;
      case mating:
         execute the mating algorithm;
         if ( mating is successful ) {
            compute endowments and update the wealths of the parents;
            update the mate pointers of each parent;
            schedule the birth event;
            cancel any parent events to occur before the birth;
         }
      case birth:
         update the wealths of the parents;
         schedule the next event occurrences for each parent;
         if ( there is an $(x, y)$ cell for the new agent ) {
            $A(t)$++;
            initialize the new agent;
            occupy the selected $(x, y)$ cell with the new agent;
            update the wealth of the new agent;
            schedule the next event occurrences for the new agent;
         }
   }
   $e$ = dequeueEvent();
}
```

Algorithm 2.3: Asynchronous time evolution

Figure 2.7 depicts a typical agent wealth time history for an asynchronous time simulation model. Note that the agent's wealth changes *linearly* over time between *random* event occurrences. If the agent's metabolic rate μ is less than regrowth rate $\rho(x, y)$ of the occupied cell, the wealth of the agent will increase between events. If μ is greater than $\rho(x, y)$, the agent's wealth will decrease. If μ is greater than $\rho(x, y)$ the agent may die. Specifically, if $w(t''') = 0$ as illustrated then death occurs at time $\tau \leq t'''$ where

$$\tau = t'' + \frac{w(t'') + r(x, y, t'')}{\mu - \rho(x, y)}. \tag{2.11}$$

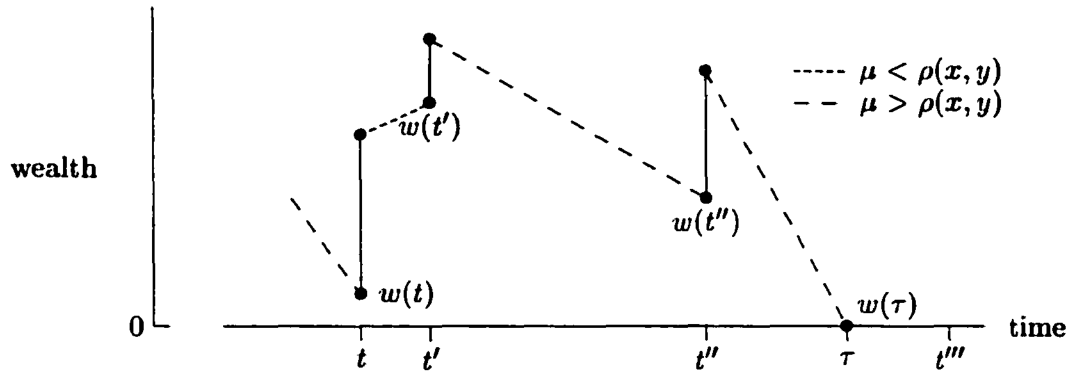


Figure 2.7: Typical agent wealth asynchronous time history

2.3 Computational Model

We now address the model at the implementation level. The class capability available in C++ naturally lends itself to implementing the artificial society model. Consequently, we utilize separate class representations for landscape cells, for the entire landscape, for agents, and for maintaining a list of future simulated events. In the following discussion, we assume the model is implemented with asynchronous time evolution using next-event simulation.

2.3.1 Implementing the Landscape

The Cell class encapsulates all states and attributes for an individual landscape cell. Included in this class are member functions for modifying the cell states; also included are functions for retrieving the values of the states and attributes. Omitting unnecessary detail[15], the basic class structure for a landscape cell follows.

```
class Cell {
  public:
    void deplete(double t);             // updates time of depletion
    void occupy(Agent* agent);          // occupies the cell
    void vacate();                      // vacates the cell

    double getCapacity(int n);          // returns nth resource capacity
    double getRegrowthRate(int n);      // returns nth regrowth rate
    double getLevel(double t, int n);   // returns nth resource level at t
    bool   isOccupied();                // returns occupancy status

  private:
    long           x, y;        // the (x,y) position of this cell
    vector<double> gamma;       // vector of N resource capacities
    vector<double> rho;         // vector of N regrowth rates
    double         t_d;         // most recent time of resource depletion
    bool           o;           // occupancy status
    Agent*         agent;       // agent occupying this cell (if any)
};
```

Consistent with the discussion of resource regrowth in Section 2.2.5.2, for each cell we store the most recent time of resource depletion rather than the current level of resource. In this way, a resource level at time t can be computed using Equation 2.9. Initially, the last time of depletion is undefined for all $(x, y) \in \mathcal{X} \times \mathcal{Y}$.

[15]In all class definitions to follow, for brevity we omit the constructor and destructor member functions with the understanding that any necessary initialization and cleanup are implemented in these functions.

Within the Landscape class, a two-dimensional data structure represents the landscape proper. A vector[16] of Cell pointers represents a single row of landscape cells. The entire landscape is then composed of a vector of landscape rows, i.e., a vector of vector data structures containing Cell pointers. The basic class structure for the landscape follows.

```
class Landscape {
  public:
    Cell* operator()(long x, long y);   // returns a pointer to (x,y) cell

  private:
    vector< vector<Cell*> > landscape;  // two-dimensional "array" of Cells
};
```

2.3.2 Implementing the Agents

The Agent class encapsulates all states and attributes for an individual agent. Included in this class are member functions that implement the agent movement, mating, birth, and death events. In addition to the states and attributes defined in Section 2.2.3, we also include the next times of occurrence of each event type for the agent. The basic class structure for an agent follows.

```
class Agent {
  public:
    void      move(Landscape* landscape);    // movement event
    void      die(Landscape* landscape);     // death event
    Agent*    mate(Landscape* landscape);    // mating event
    Agent*    deliver(Landscape* landscape); // birth event

    EventType getNextEventType();            // returns type of next event
```

[16] An object of the vector container class, defined in the C++ Standard Template Library, is treated much the same as an ordinary one-dimensional array, except that the vector class automatically manages dynamic memory allocation [1]. Nested vector data structures are treated like multi-dimensional arrays.

```
  private:
    // agent attributes
    double         phi;       // defines the agent's field of view (FOV)
    vector<double> mu;        // vector of N metabolic rates
    vector<double> w_at_beta; // vector of N initial resource wealths
    bool           sigma;     // sex of the agent {MALE, FEMALE}
    double         beta;      // time of birth
    double         lambda;    // lifespan
    double         alpha;     // age when reproduction begins
    double         omega;     // age when reproduction ends
    double         eta;       // gestation period (for females)

    // agent state variables
    long           x, y;      // current (x,y) position of the agent
    vector<double> w;         // vector of N current resource wealths
    Agent*         m_t;       // pointer to the mate (if any)

    // next times of occurrence of each event type
    double         t_move;    // time of next movement for this agent
    double         t_mate;    // time of next mating attempt
    double         t_deliver; // time of next birth event
    double         t_die;     // time of natural death
};
```

2.3.3 Implementing the List of Events

In the implementation of a next-event simulation model, a suitable data structure is chosen to store the list of future events in simulated time. The simulation engine drives the time evolution of the model by appropriately accessing events in this list. We defer further discussion of the event list until Chapter 4.

The `EventList` class encapsulates the details of the chosen event list data structure. Included in this class are member functions for accessing the event list. Omitting details, the basic class structure of the event list follows, presented here for clarity in the next section.

```
class EventList {
  public:
    void changeEvent(Agent* a);              // cancel an event and reinsert
    void deleteEvent(Agent* a);              // remove an event
    void dequeueEvent(Agent* a, double& t); // return most imminent event
    void enqueueEvent(Agent* a);             // insert an event

  private:
    // details for the implemented event list data structure
};
```

2.3.4 Implementing the Simulation

Given the maximum simulated time T and given **A_t** initialized to $A(0)$, the following code summarizes the simulation engine that drives the time evolution of the artificial society simulation model. Note this code is an implementation of Algorithm 2.3 with details of the execution of each event type contained in the corresponding **Agent** member function.

```
Landscape* landscape  = new Landscape();  // initialize the landscape
EventList* event_list = new EventList();  // initialize the event list
double     t          = 0.0;              // time parameter
Agent*     agent, mate, child;

// initialize the agents
for (long a = 0; a < A_t; a++)
  event_list->enqueueEvent(new Agent(landscape));

// get the first event in simulated time
event_list->dequeueEvent(agent, t);

while (t < T && A_t > 0) {
  switch (agent->getNextEventType()) {
    case MOVE:
      agent->move(landscape);
      event_list->enqueueEvent(agent);
      break;
    case MATE:
      mate = agent->mate(landscape);
      event_list->enqueueEvent(agent);
      if (mate)
        event_list->changeEvent(mate);
      break;

    case DELIVER:
      child = agent->deliver(landscape):
      event_list->enqueueEvent(agent);
      if (child) {
        A_t++;
        event_list->enqueueEvent(child);
      }
      break;

    case DIE:
      A_t--;
      agent->die(landscape);
      event_list->deleteEvent(Agent* a);
      delete agent;
      break;
  }
  // get the next event in simulated time
  event_list->dequeueEvent(agent, t);
}
```

Chapter 3

Artificial Society Model Output

We do not claim asynchronous time evolution to be the best approach for *all* artificial society models. In fact, we agree that synchronous time evolution may be appropriate for certain models used to simulate synchronous systems. However, many real-world systems evolve asynchronously in time and such systems are more appropriately modeled using asynchronous time evolution. Moreover, asynchronous time evolution in an artificial society model does not *guarantee* different results than synchronous time evolution, but very different behavior *may* result. Accordingly, careful consideration must be given to the time evolution of the model.

The goal of the work in [28] was to examine collective social behavior by using a discrete-event simulation model based on simple rules and initial configurations. Our focus here is on the implementation of the simulation model rather than the development of new artificial society models or the social interpretation of the results produced by these models. With this motivation, we provide results from the artificial society model defined in Chapter 2 to support our claim that, based on the choice of asynchronous or synchronous time evolution, very different behavior can be observed in the resulting output.

The experimental results to follow were motivated by an attempt to replicate the results Epstein and Axtell obtained when combining agent movement with reproduction [28]. Despite considerable experimentation, we were never able to reproduce some of the results, specifically Figure III-4 (refer to [28]) depicting large amplitude population oscillations. The authors claim that "internal dynamics alone are sufficient to generate cataclysmic events" [28]. Although we do not discount such a statement in general, we claim that, with the agent rules as defined in [28], the oscillatory behavior shown in Figure III-4 is a simulation artifact caused primarily by synchronous time evolution. Moreover, this oscillatory behavior is very much dependent upon initial conditions. To date we have been unable to determine the initial conditions used in [28] exactly.

3.1 Initial Conditions

The initial conditions for the experiments presented in this chapter are motivated by similar initial conditions defined in [28], and were determined experimentally by varying the initial conditions from [28]. For the figures presented in the sections to follow, we initialize an $X \times Y$ landscape with $N = 1$ resource. The resource regrowth rate at each (x, y) cell is $\rho(x, y) = 1.0$. Unless otherwise noted, a proportion $p = 0.16$ of the landscape cells are initially occupied by agents placed at random. The initial wealth of each of these agents is $w(0) = 10.0$. The attributes for each agent are initialized as random variates according to[1]

[1]Refer to Appendix A for a discussion of random variates.

ϕ: *Equilikely* $(1, 6)$;

μ: *Uniform* $(1.0, 4.0)$;

λ: *Uniform* $(60.0, 100.0)$;

α: *Uniform* $(12.0, 15.0)$;

ω: *Uniform* $(40.0, 50.0)$ for females, *Uniform* $(50.0, 60.0)$ for males.

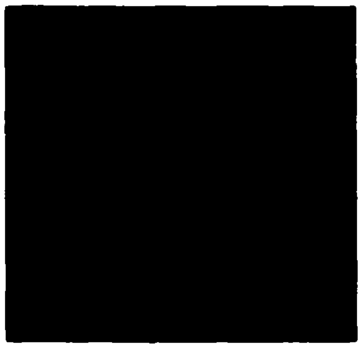
The agents execute the movement rule and the reproduction rule with $\eta = 0.0$. Unless otherwise noted, reproduction and movement are coupled, i.e., the mating event type is not modeled as a separate process. Inter-event times are assumed to be *iid Exponential* $(1/\nu)$ with $\nu = 1.0$, giving a mean inter-event time of 1.0 consistent with the integer time steps employed in [28].

3.2 Landscapes

Because Epstein and Axtell never reveal their model for the landscape resource capacity distribution (perhaps for brevity), there is some uncertainty in our choice of the distribution. Nonetheless, for our attempts to replicate results from [28], we use a two-peak Gaussian distribution that closely resembles the distribution used by Epstein and Axtell (see Figure II-1 of [28]). The resource capacity at each landscape cell is defined by Equation 2.2. A graphical depiction of our two-peak landscape is shown in Figure 3.1(b), where the brightest green corresponds to the areas of highest resource capacity and black corresponds to the areas of lowest resource capacity.

Rather than restrict the model to this two-peak Gaussian landscape, we also consider the time evolution of the model on the seven additional landscapes shown in Figure 3.1.

(a) One Peak



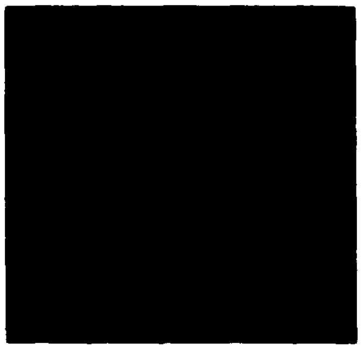
(b) Two Peaks

(c) Three Peaks

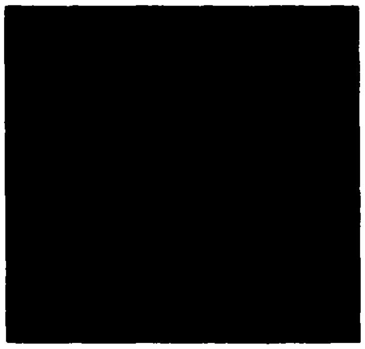
(d) Four Peaks

(e) Sixteen Peaks

(f) Random



(g) Aerial



(h) Australia

Figure 3.1: The landscapes of varying resource capacity distributions used in our experiments

Included in these figures are four additional Gaussian distributions (of one, three, four, and sixteen peaks[2]), one random distribution, and two digitized aerial photographs. In addition to varying the location of resource capacity peaks, this collection of distributions provides a representative mix of periodic boundary discontinuities. As described in Section 2.2.2, the landscape wraps around from top to bottom and from right to left, forming a torus. In landscapes with symmetric distribution (namely one, four, and sixteen peaks), an agent that moves across a landscape boundary reappears on the opposite boundary, encountering an exact reflection of the local resource capacity distribution. However, in asymmetric distributions, such as the two-peak Gaussian and Australia digital image, an agent crossing a landscape boundary is likely to move immediately from a high to a low resource capacity locale (or vice versa). In the following sections, we systematically examine output from the artificial society model using each of these landscapes.

3.3 Experimental Results

Using the initial conditions and landscapes defined in the previous two sections, we now provide results that exhibit very different behavior based solely on the choice of asynchronous or synchronous time evolution of the model. The statistic of interest that we consider here is the agent *carrying capacity* [28] of the landscape, i.e., the number of agents that the

[2]The additional Gaussian distributions in Figure 3.1(a),(c)-(e) are constructed by modifying $\gamma(x,y)$ as defined in Equation 2.2. From this equation, a single resource peak is constructed using $f(x-\delta_x X,\ y-\delta_y Y)$, where δ_x and δ_y are the x- and y-displacements of the peak from the upper left border, with $0 < \delta_x < 1$ and $0 < \delta_y < 1$. By summing together multiple $f(\cdot)$, multiple landscape peaks are achieved. For example, the three-peak distribution shown in Figure 3.1(c) is constructed using the equation

$$\gamma(x,y) = f(x-0.25X,\ y-0.25Y) + f(x-0.25X,\ y-0.75Y) + f(x-0.75X,\ y-0.5Y).$$

In addition, as more peaks are added to the landscape, the θ_x and θ_y terms in Equation 2.3 are systematically decreased to yield an appropriate mean and standard deviation of the resource capacity.

landscape is able to support. As in [28], we examine the carrying capacity as a function of simulated time. The resulting carrying capacity is most affected by the initial conditions defined in Section 3.1. For the figures presented here, we have selected a set of initial conditions that produce significantly different output for asynchronous versus synchronous time evolution of the model.

3.3.1 Basic Output for Two-Peak Landscape

For Figure 3.2, we initialize an $X \times Y = 50 \times 50$ two-peak landscape with $A(0) = 400$ agents placed at random. Depicted in the figure is the time evolution of the agent carrying capacity with $T = 2500$ for six different initial random number generator seeds[3]. Generally, synchronous time evolution (represented by the dashed lines) produces a carrying capacity that is highly oscillatory across time, with some amplitudes approaching 1500 agents, or nearly 2/3 of the maximum population. In contrast, asynchronous time evolution (represented by the solid lines) produces a much more stable carrying capacity across time. Epstein and Axtell propose explanations of the large oscillations in terms of population dynamics. However, these figures provide strong evidence that the oscillations are instead simulation artifacts resulting from the use of synchronous time evolution.

Figure 3.3 depicts the time evolution of the agent carrying capacity for an $X \times Y = 100 \times 100$ two-peak landscape with an initial $A(0) = 1600$ agents placed at random. The remaining state and attribute parameters are the same used for Figure 3.2. Again, undamped oscillatory behavior is present in the output produced using synchronous time evolution

[3]In addition to the six figures presented in Figure 3.2, similar behavior was observed using 100 other initial random number generator seeds.

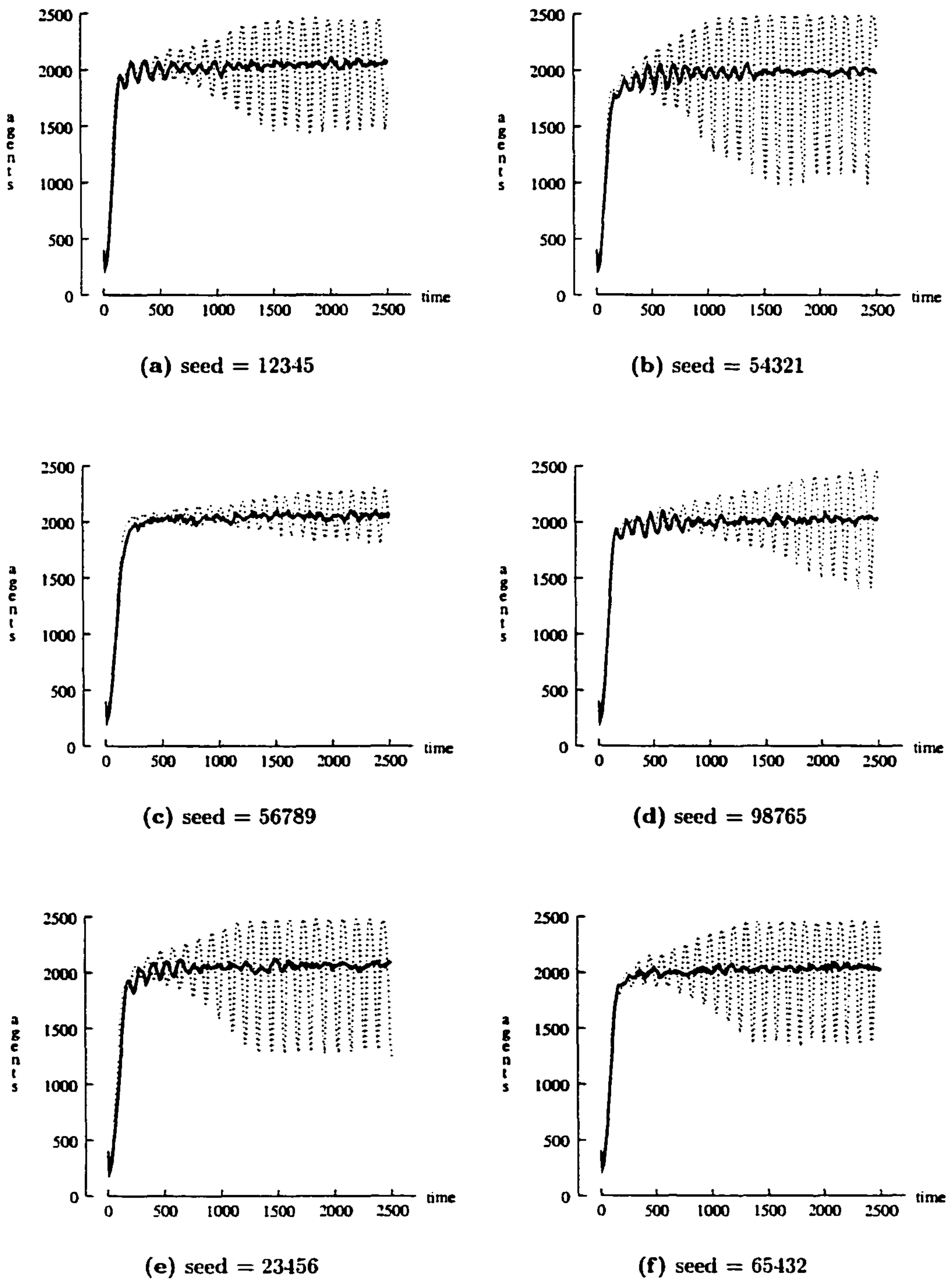
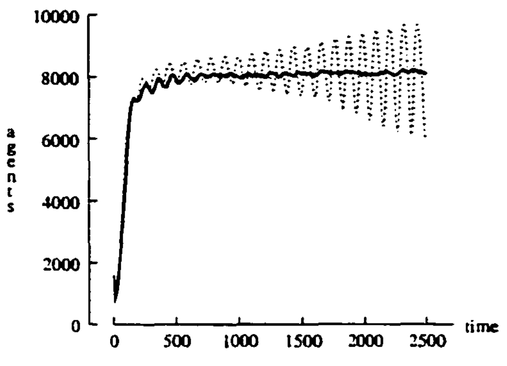
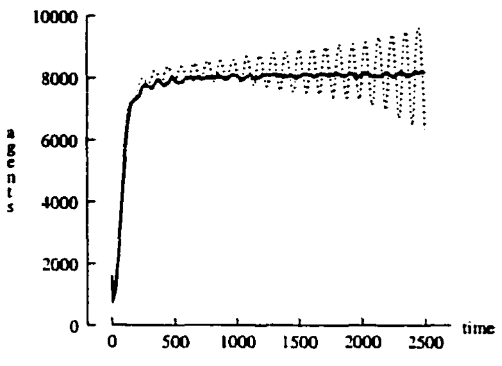


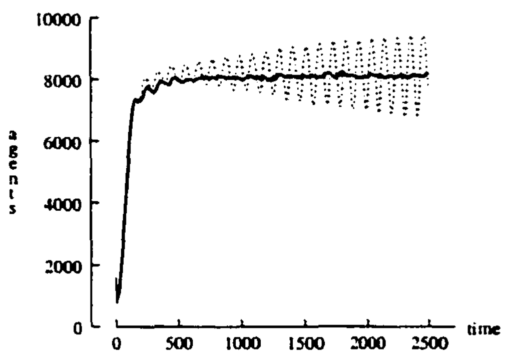
Figure 3.2: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 50×50 two-peak landscape with 0.16 of the total cells initially populated at random



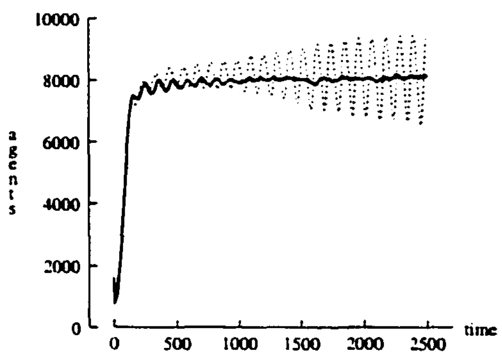
(a) seed = 12345



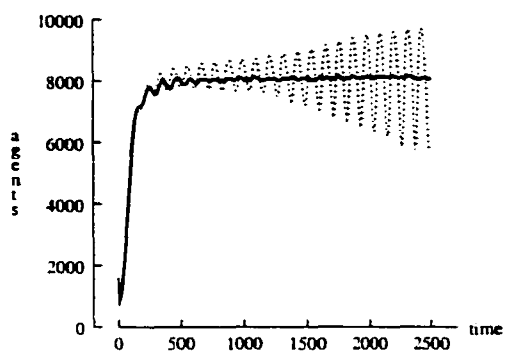
(b) seed = 54321



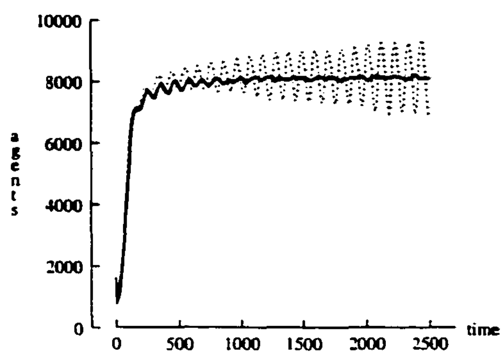
(c) seed = 56789



(d) seed = 98765



(e) seed = 23456



(f) seed = 65432

Figure 3.3: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 100×100 two-peak landscape with 0.16 of the total cells initially populated at random

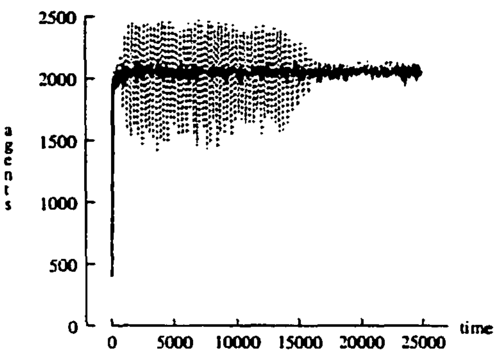
(dashed lines) but not in the output produced using asynchronous time evolution (solid lines). Although some initial oscillations can be seen in the asynchronous case, the amplitude is damped considerably compared to the synchronous case, and the oscillations are quickly suppressed. Moreover, changes in the output from one initial seed to another are far less dramatic using asynchronous time evolution. That is, unlike the synchronous results, the asynchronous results are not sensitive to the random variate sequence of movement and reproduction, as manifested by the choice of initial seed.

3.3.2 Output for Two-Peak Landscape with Large T

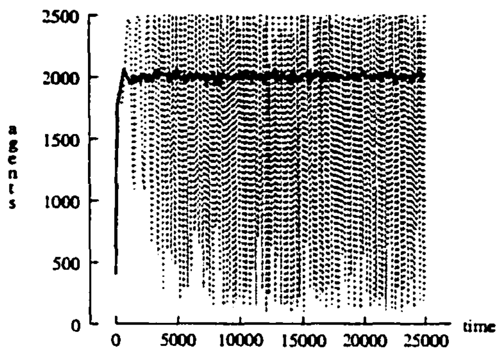
We feel that $T = 2500$, as suggested by Epstein and Axtell, may not yield a sufficient measure of the long-term time evolution of the agent carrying capacity. For this reason, Figure 3.4 depicts the time evolution of the carrying capacity with the same initial conditions and 50×50 two-peak landscape as Figure 3.2, but with $T = 25\,000$. In general, the highly oscillatory behavior in the output produced by synchronous time evolution persists, while asynchronous time evolution produces a stable carrying capacity[4]. Notice in Figure 3.4(a) that the oscillatory behavior in the synchronous case is eventually suppressed, but not until after $T = 15\,000$. In Figure 3.4(e), the oscillatory behavior is suppressed even more quickly, but reappears around $T = 22\,500$. For the remaining four seeds, the oscillatory behavior persists for the lifetime of the model.

Similarly, Figure 3.5 depicts the time evolution of the agent carrying capacity for the same initial conditions and 100×100 two-peak landscape as Figure 3.3, but with $T = 25\,000$. Again, oscillatory behavior persists in the output produced by synchronous time evolution.

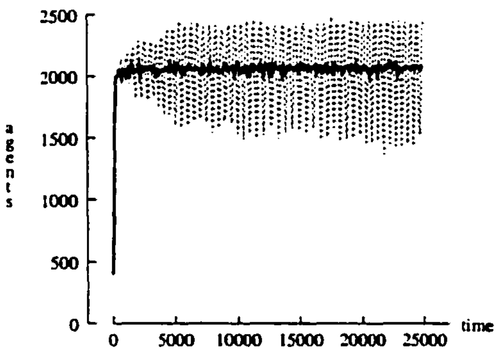
[4]Similar behavior was observed using 100 other initial seeds.



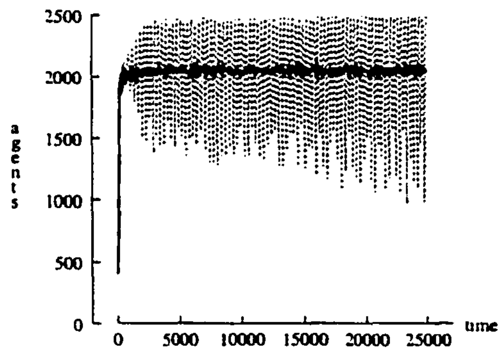
(a) seed = 12345



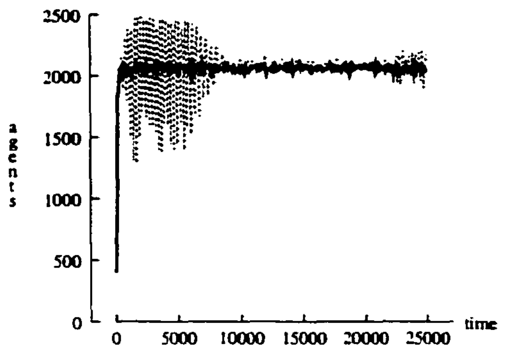
(b) seed = 54321



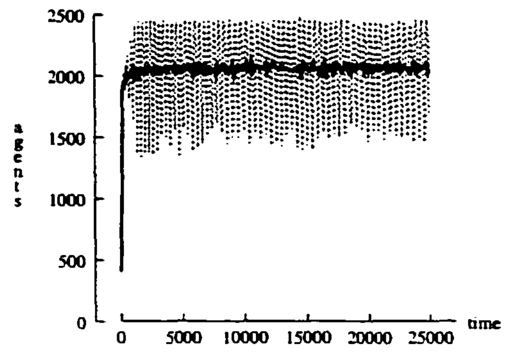
(c) seed = 56789



(d) seed = 98765

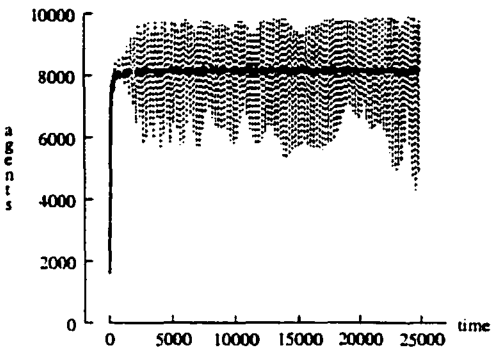


(e) seed = 23456

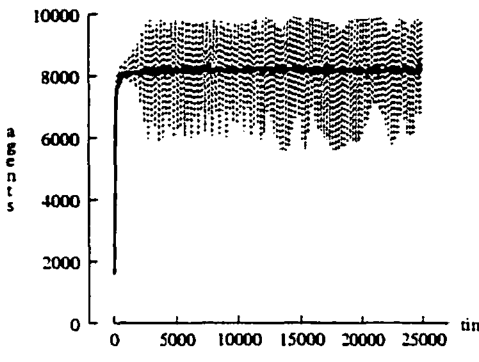


(f) seed = 65432

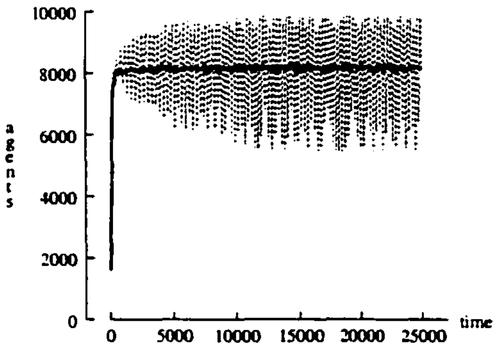
Figure 3.4: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 25\,000$ on a 50×50 two-peak landscape with 0.16 of the total cells initially populated at random



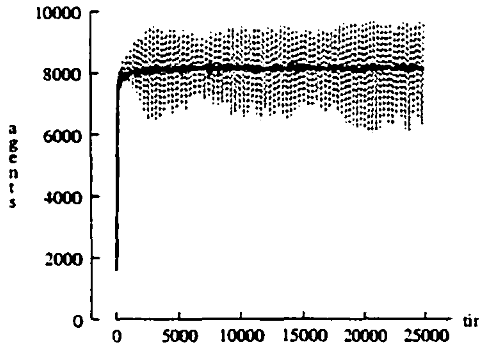
(a) seed = 12345



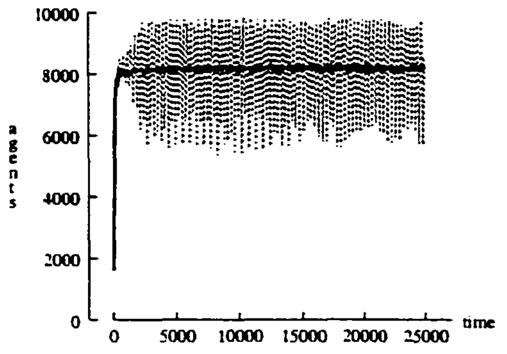
(b) seed = 54321



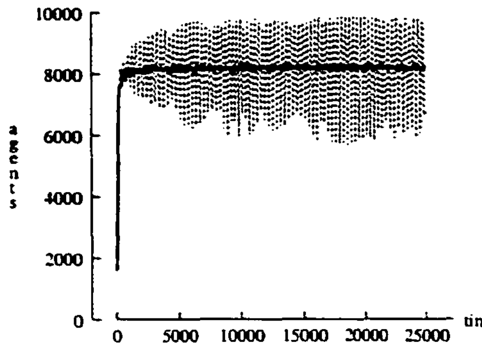
(c) seed = 56789



(d) seed = 98765



(e) seed = 23456



(f) seed = 65432

Figure 3.5: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 25\,000$ on a 100×100 two-peak landscape with 0.16 of the total cells initially populated at random

In contrast, asynchronous time evolution quickly produces a stable agent carrying capacity[5].

3.3.3 Output for Two-Peak Landscape with Large $A(0)$

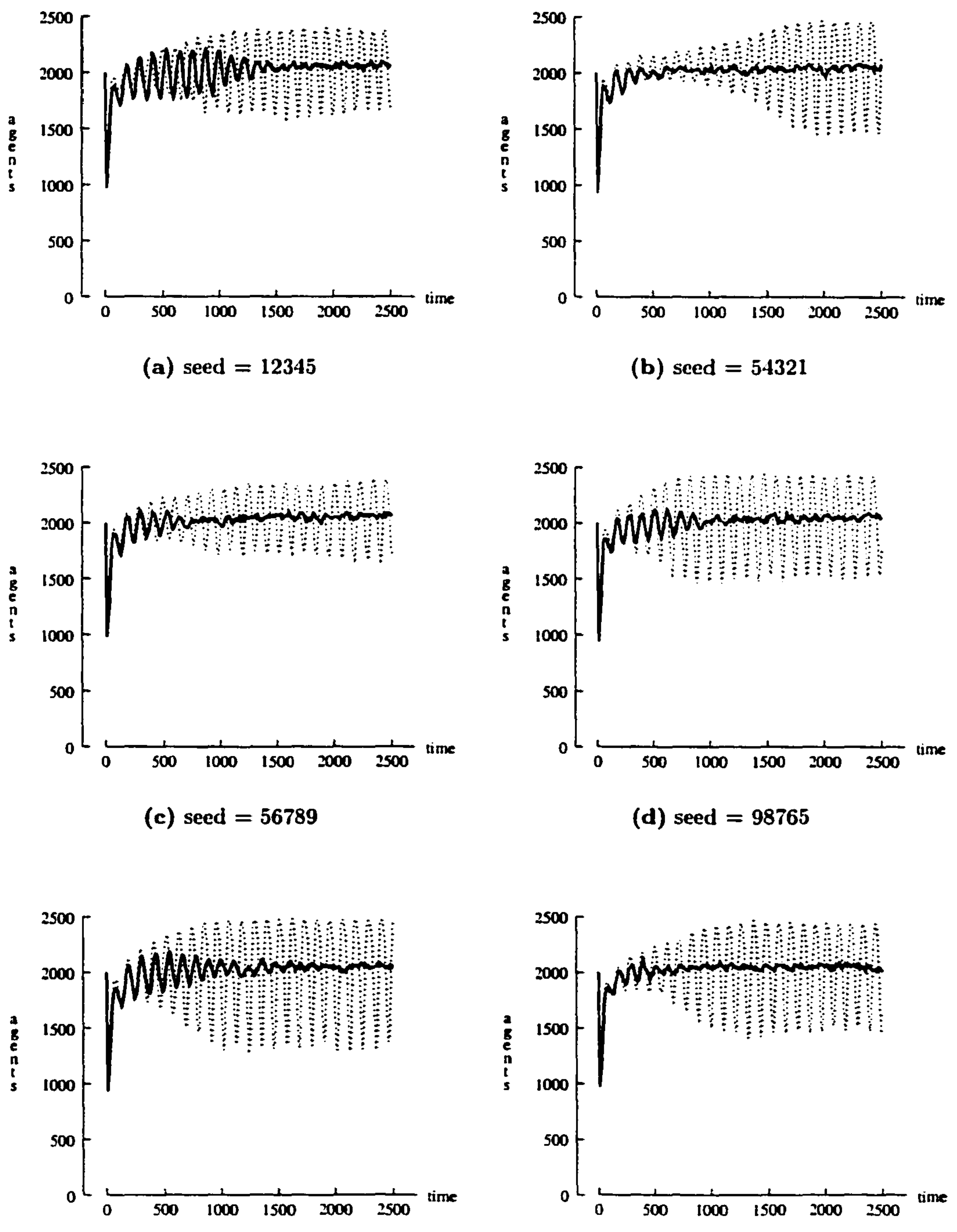
Furthermore, we choose not to restrict our results to a model with only $p = 0.16$ of the landscape cells initially populated. We consider an initial population determined by the long-term asynchronous time evolution carrying capacity achieved when $p = 0.16$ of the total cells are initially populated at random. In Figure 3.6, we initialize a 50×50 two-peak landscape with $A(0) = 2000$ agents placed at random. In this figure, synchronous time evolution again produces highly oscillatory behavior. Asynchronous time evolution also produces some initial oscillations[6] (although typically much smaller in amplitude than synchronous time evolution) but the oscillations are eventually suppressed, yielding a stable agent carrying capacity. Also note that asynchronous time evolution produces the same long-term carrying capacity for $A(0) = 400$ agents (Figure 3.2) and for $A(0) = 2000$ agents (Figure 3.6). Similar behavior was observed using a 100×100 two-peak landscape initialized with $A(0) = 1600$ agents and $A(0) = 8000$ agents.

3.3.4 Output for Large Dimension Two-Peak Landscape

We also consider the time evolution of the carrying capacity on two-peak landscapes of increasing dimensions, namely $X \times Y$ with $X = Y$ and $X \in \{150, 200, 250, 300, 350, 400, 450, 500, 1000\}$, each with $p = 0.16$ of the total cells initially populated at random. For

[5]Once again, similar behavior was observed using 100 other initial seeds.

[6]Despite considerable experimentation (including agent state/attribute distributions constructed using statistics gathered over many time steps), we were unable to determine a set of initial conditions that produced an initially stable carrying capacity. That is, we were able to reduce, but not eliminate, the initial drop in agent carrying capacity.



(a) seed = 12345

(b) seed = 54321

(c) seed = 56789

(d) seed = 98765

(e) seed = 23456

(f) seed = 65432

Figure 3.6: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on a 50×50 two-peak landscape with initial population of 2000 agents placed at random

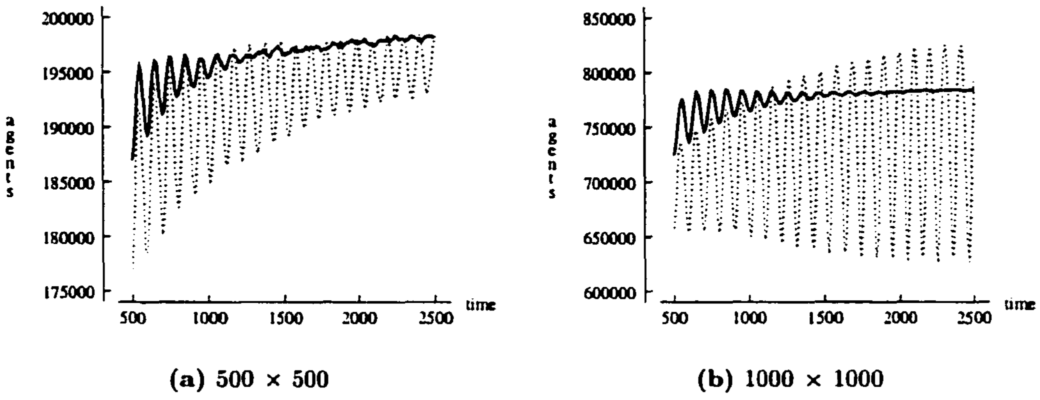
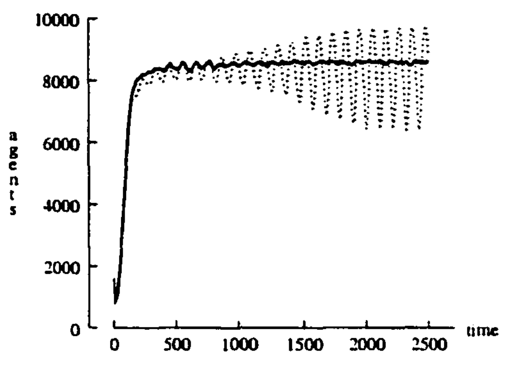


Figure 3.7: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ on 500×500 and 1000×1000 two-peak landscapes with 0.16 of the total cells initially populated at random

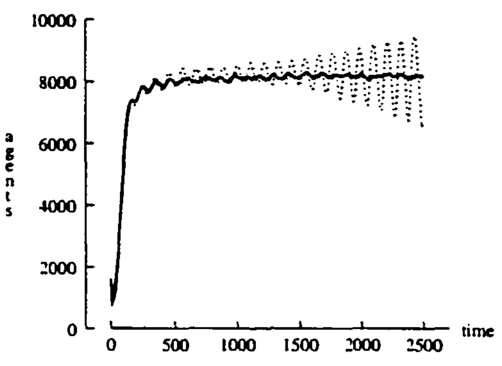
brevity, we present in Figure 3.7 only those results for $X \times Y = 500 \times 500$ and $X \times Y = 1000 \times 1000$. In both figures, we restrict the horizontal and vertical scales to emphasize the differences in the output produced by asynchronous and synchronous time evolution. Similar to the output for smaller dimensions, synchronous time evolution produces large oscillations in the carrying capacity, while asynchronous time evolution produces a stable carrying capacity (after small initial oscillations). Also note for these larger dimensions that the long-term carrying capacity produced by asynchronous time evolution no longer tracks the mean of the oscillations produced by synchronous time evolution, as observed for the smaller dimensions. That is, the long-term *mean* agent population is significantly higher for asynchronous than for synchronous time evolution on the larger landscapes.

3.3.5 Output using Other Landscapes

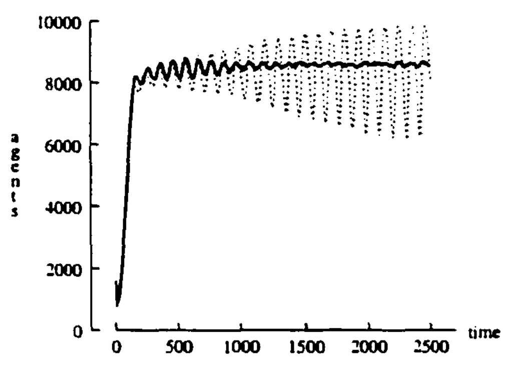
To show that these different behaviors are not confined to the two-peak landscape, we now consider the time evolution of the agent carrying capacity on the seven other resource



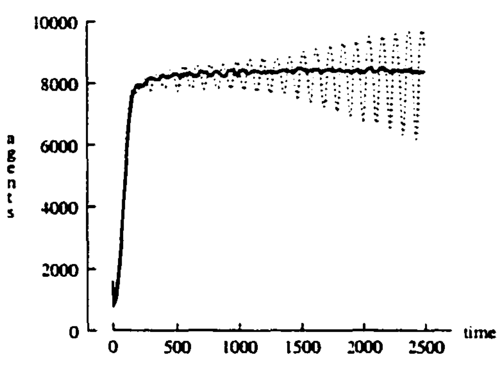
(a) One Peak



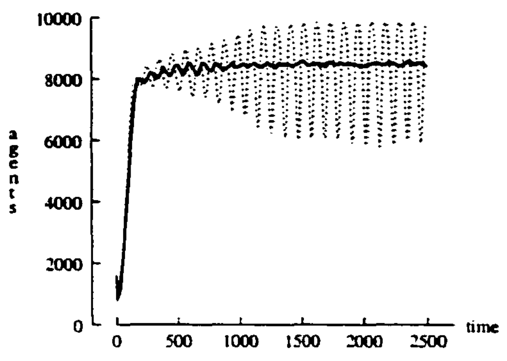
(b) Three Peaks



(c) Four Peaks



(d) Sixteen Peaks



(e) Random

Figure 3.8: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions with 0.16 of the total cells initially populated at random

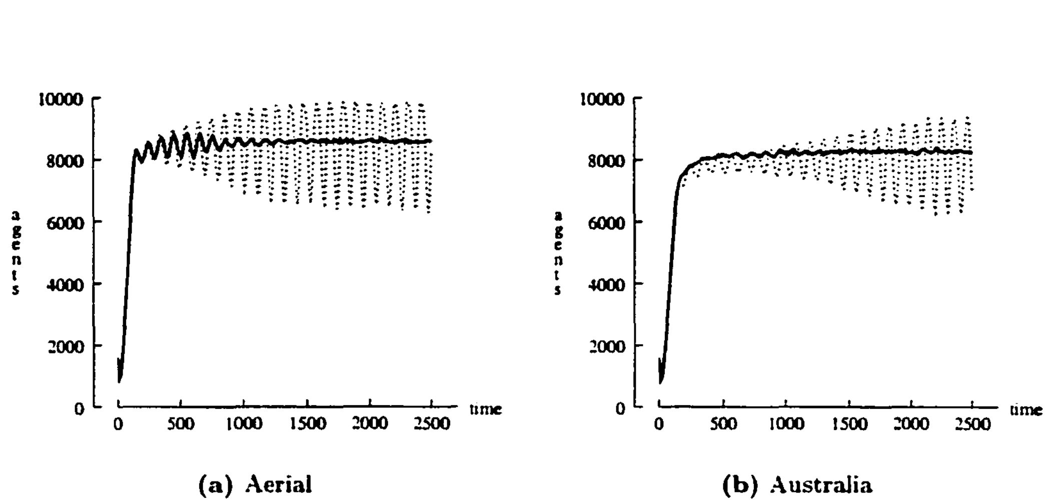
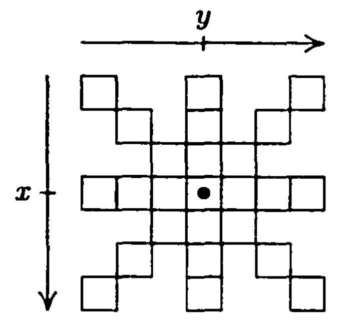
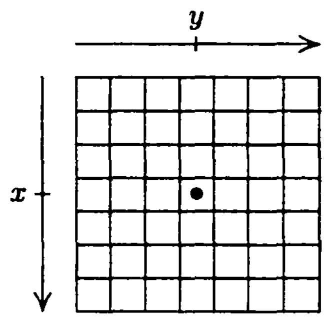


Figure 3.9: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for T = 2500 and initial seed 12345 on 100 × 100 digital image landscapes with 0.16 of the total cells initially populated at random

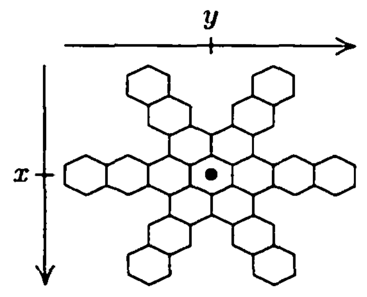
capacity distributions given in Figure 3.1. Figure 3.8 depicts the carrying capacity for the five additional constructed resource capacity distributions, i.e., the four Gaussian and the random distributions. Each figure was generated with $T = 2500$ using a 100 × 100 landscape and $p = 0.16$ of the total cells initially populated at random. As with the two-peak landscape, synchronous time evolution produces large oscillations while asynchronous time evolution produces a stable carrying capacity. Figure 3.9 depicts similar results using the two digital photographs for the resource capacity distributions. Figures 3.8 and 3.9 suggest that the different behaviors present in synchronous and asynchronous output are independent of the choice of underlying resource capacity distribution.



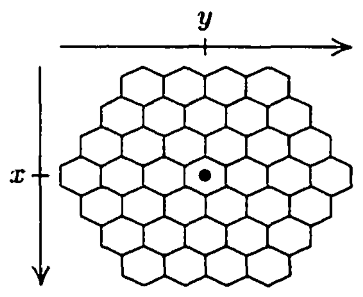
(a) Eight-directional FOV



(b) Moore FOV



(c) Six-directional hex FOV



(d) Complete hex FOV

Figure 3.10: Alternate Fields of View (FOV) for an agent at (x, y) if $\phi = 3$

3.3.6 Output using Alternate Fields of View and Cell Shape[7]

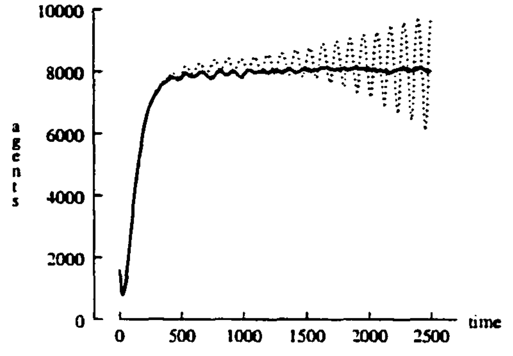
To further show that different output can be achieved based solely on the choice of asynchronous or synchronous time evolution, we consider two alternate fields of view (FOV). In addition to the four-directional FOV depicted in Figure 2.3, we consider model output using an eight-directional FOV, depicted in Figure 3.10(a), and a Moore neighborhood FOV, depicted in Figure 3.10(b). Using the eight-directional FOV, an agent with field of view attribute ϕ located at cell (x, y) can see ϕ cells from (x, y) in each of the eight directions north, northeast, east, southeast, south, southwest, west, and northwest. Using the Moore neighborhood FOV, an agent with field of view attribute ϕ located at (x, y) can see any cell

[7]Thanks to Bryan Walter for his assistance in developing the code for this section during a research project for UMSA, an undergraduate research course at William and Mary.

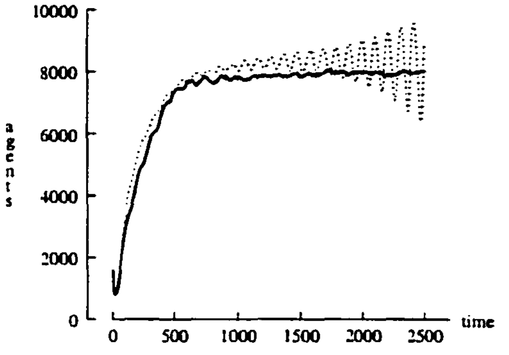
in the $(2\phi+1)\times(2\phi+1)$ square of cells centered at (x,y)[8]. Using either the eight-directional or Moore neighborhood FOV, an agent attempting to reproduce will consider as a potential mate any fertile agent of the opposite sex located in one of the *eight* nearest neighbor cells.

Furthermore, we consider model output using hexagonal shaped cells, rather than square cells, and two associated fields of view. With this modification, we consider a six-directional hexagonal FOV, depicted in Figure 3.10(c), and a complete hexagonal FOV, depicted in Figure 3.10(d). Using the six-directional hexagonal FOV, an agent with field of view attribute ϕ located at cell (x,y) can see ϕ cells from (x,y) in each of the six directions west, northwest, northeast, east, southeast, and southwest. Using the complete hexagonal FOV, an agent with field of view attribute ϕ located at cell (x,y) can see any cell in the hexagon consisting of $1+3\phi(\phi+1)$ cells centered at (x,y)[9]. Using either of these hexagonal FOVs, an agent attempting to reproduce will consider as a potential mate any fertile agent of the opposite sex located in one of the *six* nearest neighbor cells.

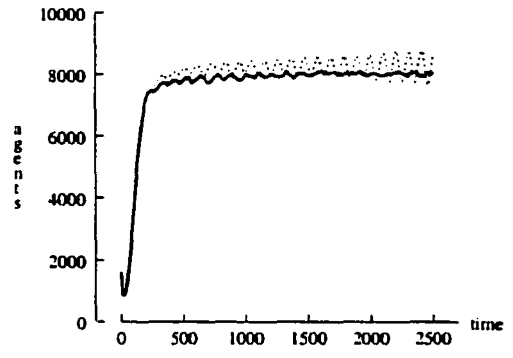
Figure 3.11 depicts the carrying capacity for $T = 2500$ using a 100×100 two-peak landscape with $p = 0.16$ of the total cells initially populated at random. The remaining initial conditions are the same as described in Section 3.1. Again, oscillatory behavior is present in the output produced using synchronous time evolution, while asynchronous time evolution produces a stable carrying capacity. This figure provides strong evidence that the differences in output are not biased by the choice for agent field of view.



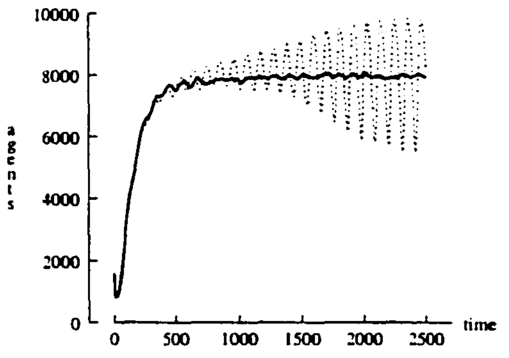
(a) Eight-directional FOV



(b) Moore FOV



(c) Six-directional hex FOV



(d) Complete hex FOV

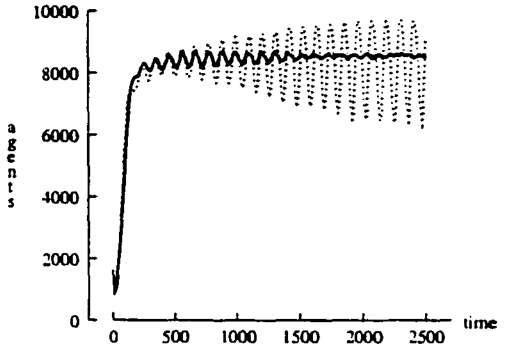
Figure 3.11: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity using alternate agent fields of view (FOV) for $T = 2500$ and initial seed 12345 on 100×100 two-peak landscape with 0.16 of the total cells initially populated at random

3.3.7 Output using Non-Periodic Landscape Boundary Conditions

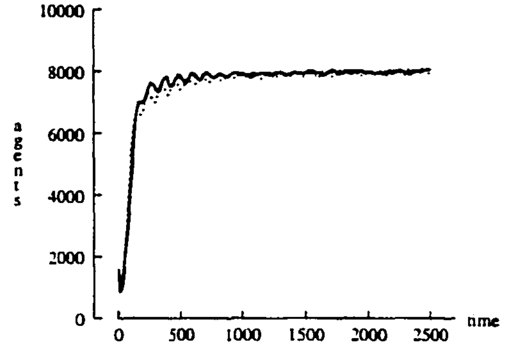
We also consider the effect on output of removing the periodic landscape boundary conditions. That is, the landscape no longer forms a torus, but rather the FOV of an agent near the landscape border is reduced in size because of the agent's proximity to the border.

[8]Note that by permitting diagonal movement, any cell in the square of cells centered at (x, y) is at most ϕ cells away from (x, y).

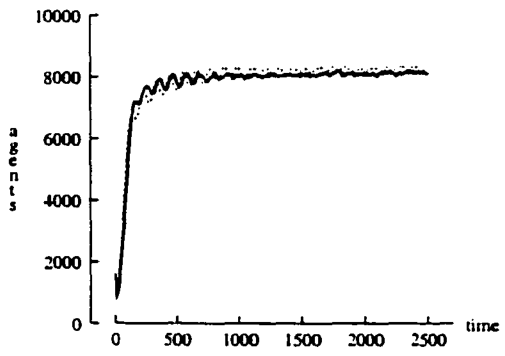
[9]Similar to the Moore neighborhood FOV, by permitting diagonal movement any cell in the hexagon is at most ϕ cells away from (x, y)



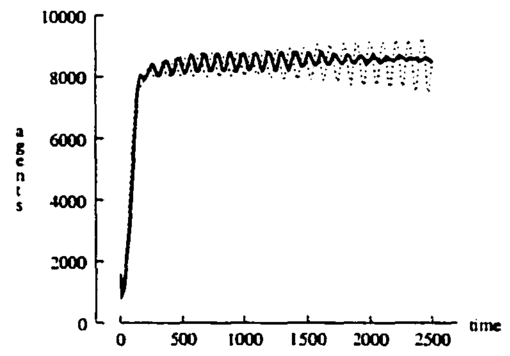
(a) One Peak



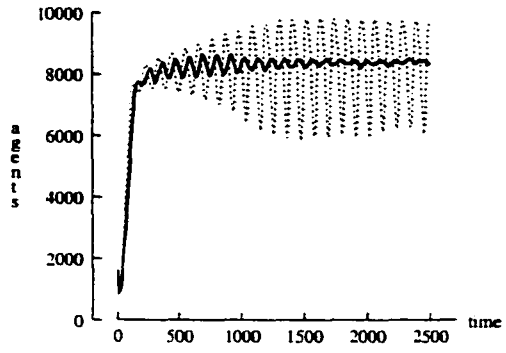
(b) Two Peaks



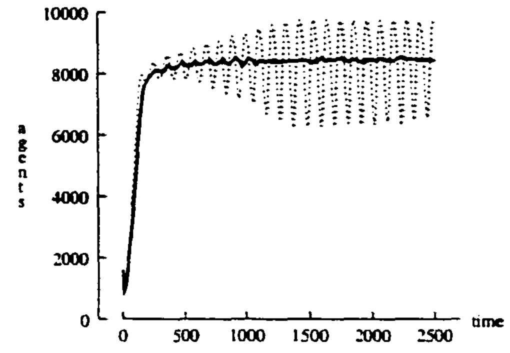
(c) Three Peaks



(d) Four Peaks



(e) Sixteen Peaks



(f) Random

Figure 3.12: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity with non-periodic boundary conditions for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions and 0.16 of the total cells initially populated at random

Figure 3.12 depicts the agent carrying capacity using non-periodic boundary conditions on 100×100 landscapes with different resource capacity distributions. For each figure, $T = 2500$ and $p = 0.16$ of the total cells are initially populated at random. Note that the output produced using asynchronous time evolution is essentially unaffected by the use of non-periodic boundary conditions, i.e., as with periodic boundary conditions, a stable carrying capacity results after small initial oscillations. However, the output produced using synchronous time evolution is dependent on the underlying resource capacity distribution when using non-periodic boundary conditions. The output produced using synchronous time evolution and landscape distributions that have no large discontinuities at the borders (Figure 3.12(a),(d)-(f)) contains large oscillations in the carrying capacity; output produced using landscape distributions that have dramatic border discontinuities (Figure 3.12(b) and (c)) do not contain large oscillations. These figures suggest that, unlike asynchronous time evolution, the output produced using synchronous time evolution is sensitive to the underlying resource capacity distribution when non-periodic boundary conditions are employed.

3.3.8 Output with Movement and Reproduction Uncoupled

Finally, we consider the effects on output that result from uncoupling agent movement and reproduction under asynchronous time evolution. As described in Section 2.2.5.2, when using asynchronous time evolution the mating event type can either be coupled with the movement event type or modeled as a separate stationary Poisson process. Unlike all previous results presented, here we assume that the two event types are uncoupled, each with inter-event times that are *iid* $Exponential(1/\nu)$ with $\nu = 1.0$.

First we consider output that results from uncoupling agent movement and reproduc-

Landscape	Std Dev 70	Std Dev 80
One Peak	0.35	0.01
Two Peaks	0.24	0.00
Three Peaks	0.20	0.00
Four Peaks	0.37	0.10
Sixteen Peaks	0.46	0.11
Random	0.19	0.00
Aerial	0.50	0.03
Australia	0.25	0.00

Table 3.1: Proportion of replications that exhibit oscillatory behavior using asynchronous time evolution with agent movement and reproduction uncoupled, and gestation period $\eta = 0.0$. All landscapes have common resource mean of 128.

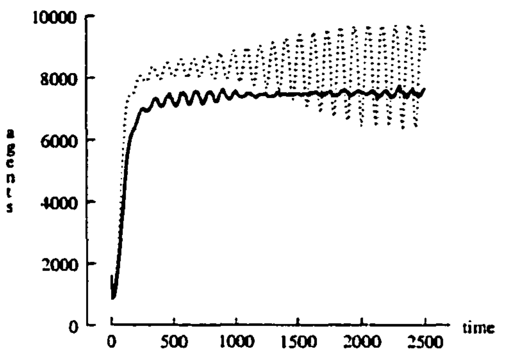
tion, with gestation period $\eta = 0.0$. For each of the seven landscapes, Table 3.1 shows the proportion of 100 replications in which large oscillatory behavior appears using asynchronous time evolution with movement and reproduction uncoupled and $\eta = 0.0$. For each landscape, we consider two different resource capacity distributions (each with resource capacity mean of 128): one with a resource capacity distribution standard deviation of 70, and one with standard deviation of 80. For each of the 100 replications, we use a 100×100 landscape with $p = 0.16$ of the total cells initially populated at random, and with the remaining initial conditions as defined in Section 3.1.

In this case, as shown in Table 3.1 the likelihood that oscillatory behavior occurs using asynchronous time evolution is dependent on the standard deviation of the underlying resource capacity distribution. As shown in the table, a resource capacity standard deviation of 70 results in a significantly higher proportion of replications that exhibit large oscillatory behavior using asynchronous time evolution, while a standard deviation of 80 results in very few replications that exhibit such behavior[10]. Note that, consistent with the results

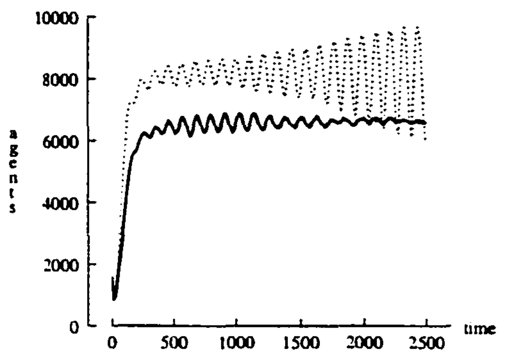
[10]Standard deviations less than 70 produce roughly the same proportions as a standard deviation of 70.

presented earlier in this chapter, results produced using asynchronous time evolution with movement and reproduction *coupled* are *not* sensitive to the standard deviation of the underlying resource capacity, i.e., a stable carrying capacity is consistently produced. Also note that synchronous time evolution produces highly oscillatory behavior regardless of the standard deviation of the underlying resource capacity distribution. Moreover, by uncoupling movement and reproduction using asynchronous time evolution and by adjusting the standard deviation of the resource capacity distribution, we are able to mimic the highly oscillatory behavior produced by synchronous time evolution, but only by significantly altering the underlying event sequence.

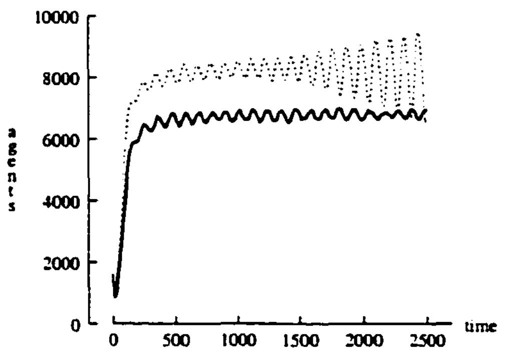
We also consider the effect of uncoupling agent movement and reproduction, but with gestation period $\eta = 1.0$. As depicted in Figure 3.13, by incorporating a nonzero gestation period, the mean agent carrying capacity is lowered significantly compared to the output from synchronous time evolution. Note that, in this case, asynchronous time evolution consistently produces output with persistent small oscillations, but with the amplitude of oscillations reduced dramatically compared to synchronous time evolution.



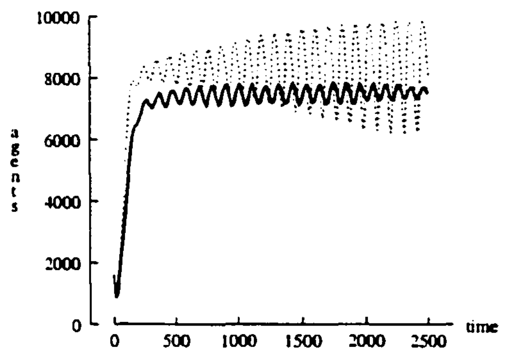
(a) One Peak



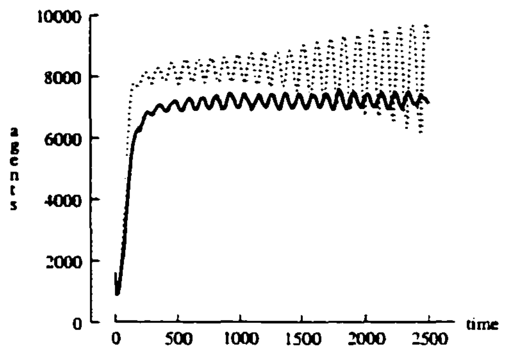
(b) Two Peaks



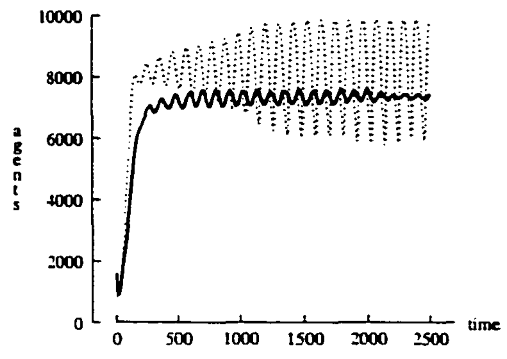
(c) Three Peaks



(d) Four Peaks



(e) Sixteen Peaks



(f) Random

Figure 3.13: Asynchronous (——) vs. synchronous (- - -) time evolution of agent carrying capacity for $T = 2500$ and initial seed 12345 on 100×100 landscapes of different resource distributions with 0.16 of the total cells initially populated at random. For asynchronous time evolution, agent movement and mating are uncoupled, with gestation period $\eta = 1.0$.

3.3.9 Output Summary

Again, we reemphasize that we are not promoting asynchronous time evolution as the best approach for *all* artificial society models, but we present these results as evidence that very different output can result from modeling time synchronously or asynchronously. We have shown that different output is independent of the landscape size, the underlying landscape distribution, the maximum time of the simulated model, the initial agent population, the agent field of view, and the landscape cell shape. Moreover, unlike synchronous time evolution, asynchronous time evolution is not sensitive to the landscape boundary conditions, nor to the random sequence of events as manifested by the choice of initial seed. In addition, asynchronous time evolution provides more flexibility to the model by permitting easy changes to complex behavioral rules, such as the separation of agent movement and reproduction and the incorporation of a gestation period. In summary, we claim that serious consideration must be given to the time evolution of the artificial society model, and we offer asynchronous time evolution as an attractive alternative for those models involving natural asynchronous behavior.

Chapter 4

The Event List

A discrete-event simulation model requires the use of a time-advance mechanism to ensure that simulated events occur in the correct order. In a next-event simulation, *next-event* time advance is typically coupled with event scheduling [82]. The next occurrence for each possible event type is scheduled for execution at some simulated time in the future. The time-advance mechanism repeatedly determines the most imminent possible event in the schedule and advances the simulation clock to this event's scheduled time of occurrence. The data structure that represents the schedule of events to occur in simulated time is called an *event list*.

Although asynchronous time evolution can reduce simulation artifacts in an artificial society discrete-event simulation model, computational performance can suffer dramatically unless the event list is implemented properly. In this chapter we first provide background discussion on previous event list research. We then introduce select standard and novel data structures and associated algorithms for implementing the event list in our model. Finally we demonstrate that acceptable computational performance can be achieved by a judicious choice of the event list implementation.

4.1 Background

In the literature, comparative studies of event list data structures attempt to generalize for use in many applications [56, 67, 91]. This type of generalization provides basic guidelines for selecting an event list implementation, but does not determine the best implementation for a specific application. Indeed, no single implementation is best in all applications [56].

The most widely used approach for measuring event list performance in a discrete-event simulation is the classic hold model [32, 47, 48, 55, 56, 67, 71]. In the hold model, the set of pending events is represented by a priority queue in which the priority of each event is the time that event is to occur. The execution of an event causes exactly one new event to be scheduled, thereby ensuring an event list of constant size. A hold operation consists of an *enqueue*, which schedules an event at some future time, and a *dequeue*, which finds and returns the most imminent event in the priority queue. Performance is measured using the average time required for a hold operation as a function of queue size.

Even in the simplest form of our artificial society model, i.e., with agent movement but no reproduction, the associated event list does not conform to the hold model. As time evolves, agent mortality produces an event list that is not of constant size. With agent reproduction included in the model, the execution of a single event can cause more than one new event to be scheduled. Hence, performance measurement within the hold model provides insight, but an investigation of the performance of event list implementations within our artificial society model is required.

4.2 Event List Implementations

Our goal is not an exhaustive comparison of available event list data structures. Such work exists in the framework of the hold model [56, 67, 91]. Instead our goal is to compare the performance of selected event list implementations in our representative artificial society model. Our selections are based on the results of these previous event list comparisons and the corresponding suggestions by the authors. We choose to examine event list implementations that perform consistently well in the hold model, are of general interest, and/or are best suited for additional event list operations such as arbitrary deletion and priority changes.

There are three basic types of event list structures: list-based structures (e.g., linked list), tree-based structures (e.g., heaps, binary trees), and multiple-list structures (e.g., two-list). For this study, we consider one representative structure from each of the three basic types based on results and suggestions by other authors. In addition, because we want identical output regardless of the event list implementation, we consider only *stable* implementations. A stable event list implementation is one in which events with identical time stamps are always processed in FIFO order [56].

Based on the event types defined in Section 2.2.5.2, there are three event list operations to consider[1] when evaluating an event list implementation: *dequeue*, *enqueue*, and *cancel*[2]. A *dequeue* operation finds and returns the most imminent event in the event list. An *enqueue* operation schedules an event by inserting the event in the appropriate position in

[1]As defined in Section 2.3.3, the `EventList` class contains four member functions. We do not consider `deleteEvent(·)` here because, in the context of the artificial society simulation model, the delete operation consists only of freeing the memory of a dequeued event.

[2]Note the `changeEvent(·)` member function defined Section 2.3.3 is composed of a cancel operation (to be described) followed by a subsequent enqueue operation.

the event list. A *cancel* operation finds and removes a designated event from the event list.

Before we introduce the selected event list implementations, consider the components of a typical event. In our artificial society model, each element in the event list should contain the time that an event is to occur, the type of that event, and a pointer to the corresponding agent. Given that each agent can execute only one type of event at a time, we simplify the event list by including only the most imminent event for each agent. In this way, there is one event per agent in the event list. For each agent, the type of the most imminent event can be determined from the agent data structure, as given in Section 2.3.2. Hence, an element in the event list contains only the time of the event and a pointer to the associated agent. This event structure is used throughout the remainder of this chapter.

4.2.1 Sorted Singly-Linked List

Easy to implement and understand, a singly-linked list data structure is the natural first approach for representing the event list. If we sort the events by increasing time of occurrence, we obtain a sample event list like the one shown in Figure 4.1.

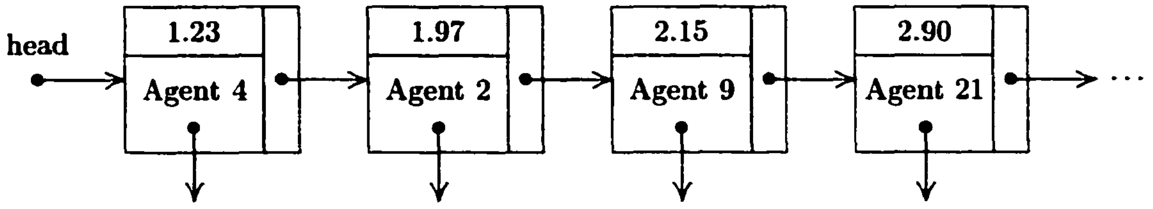


Figure 4.1: A sample singly-linked list event list

Using this approach, the next event to occur in simulated time is always the first element in the linked list. In this way, a dequeue operation simply returns the first element, i.e., the head, of the linked list; the time complexity of a dequeue operation is therefore constant.

For an enqueue operation, a linear search ensues (starting from the head) to determine the appropriate insertion point. Similarly, a cancel operation requires a linear search. Correspondingly, the time complexities of the enqueue and cancel operations are linear. For a large number of events, i.e., as the number of agents grows, these linear searches seriously degrade the computational performance of the simulation. The time complexities of the three event list operations using a sorted singly-linked list are summarized in the following table.

Operation	Complexity
dequeue	$O(1)$
enqueue	$O(A(t))$
cancel	$O(A(t))$

Table 4.1: Time complexities for sorted linked list

4.2.2 Henriksen's Algorithm

Henriksen's algorithm [47, 48], a list-based structure, involves a clever modification to the sorted linked list approach and is reported to perform in practice as well as any tree-based priority queue structure [56, 67]. The linked list in Henriksen's approach is sorted by decreasing event time, as shown in Figure 4.2. To facilitate easy insertions and deletions, the list includes minimum and maximum time sentinels at the ends. Because the next event to occur in simulated time is always the next-to-last element in Henriksen's linked list, a dequeue operation remains a constant time operation, as in the sorted linked list.

For an enqueue or cancel operation, complete linear searches of the linked list are avoided by using an auxiliary search array, also depicted in Figure 4.2. Each element in this search array consists of a scheduled event time and a pointer to the associated event in the linked list. The search array contains sample times from the event list; these samples are sorted by decreasing event time. A standard binary search of the search array is used to find the smallest time in the search array greater than the event time sought. Beginning with the event in the linked list pointed to by that search array element, a descending linear search of the linked list ensues until the desired event time is found or it is determined that the event time is not in the list.

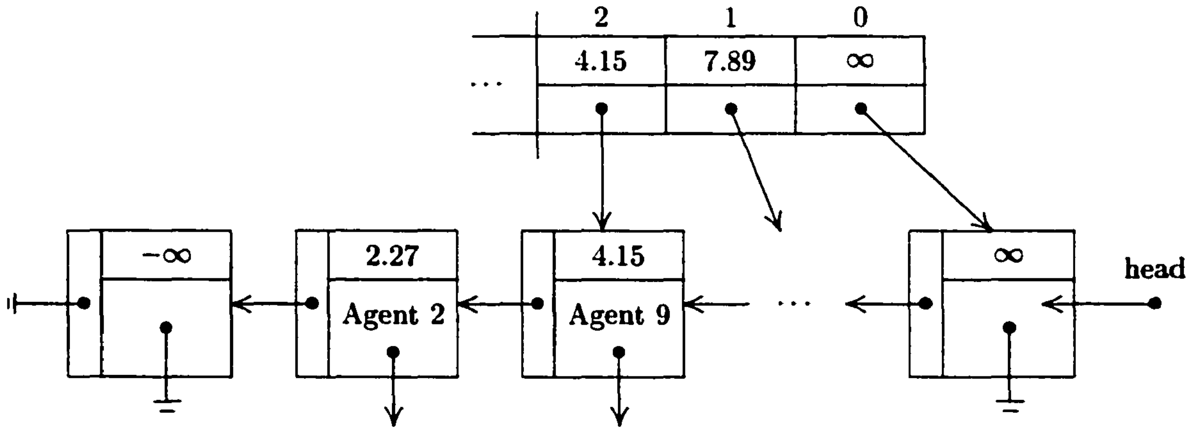


Figure 4.2: A sample Henriksen's event list

During the linear search of the linked list, Henriksen's "pull" technique [48] attempts to keep the search array pointers evenly distributed throughout the linked list. While descending the linked list, each κth event encountered is placed into the next successive element in the search array[3], where κ is a predefined constant[4]. The goal of the "pull" technique is to reduce the length of the linear search through the linked list. Using Henriksen's algorithm, the average and amortized cost of an insertion in the hold model is sublinear [59]. More

[3]We initially allocate a dynamic search array of size 1024 elements. The end of the search array, determined by the number of valid entries in the array, is maintained. When necessary, the size of the search array is systematically doubled.

[4]We use the same $\kappa = 4$ defined by Henriksen [48].

specifically, the amortized cost is often $O(\log(A(t)))$ and is limited by $O(\sqrt{A(t)})$ in the worst case [91]. The time complexities of the three event list operations using Henriksen's algorithm are summarized in the following table.

Operation	**Complexity**
dequeue	$O(1)$
enqueue	$O\left(\sqrt{A(t)}\right)$ (amortized)
cancel	$O\left(\sqrt{A(t)}\right)$ (amortized)

Table 4.2: Time complexities for Henriksen's algorithm

4.2.3 Splay Tree

The splay tree, originally developed by Sleator and Tarjan [100] and shown to be the most efficient tree-based priority queue implementation in practice[5] [56, 67], is a form of binary search tree (BST) that uses *splay* operations to maintain a balanced tree. Tree operations, such as inserts, deletes, and joins, are performed in the same manner as in a typical BST, but are followed by a splay operation. A sample splay tree implementation of the event list is depicted in Figure 4.3.

A splay operation consists of a sequence of rotations that brings the most recently accessed node to the root of the tree. At the same time, the splay brings other nodes encountered on the search path closer to the root. More specifically, the splay rotations halve the distance from the root to any node encountered on the search path [98].

A splay at node r is defined by the following steps [50].

[5]Within a variation of the hold model, Marín claims the complete binary tree (CBT) is the most efficient tree-based priority queue implementation [67]. Despite multiple requests, we were unable to obtain a copy of the corresponding technical report referenced in [67].

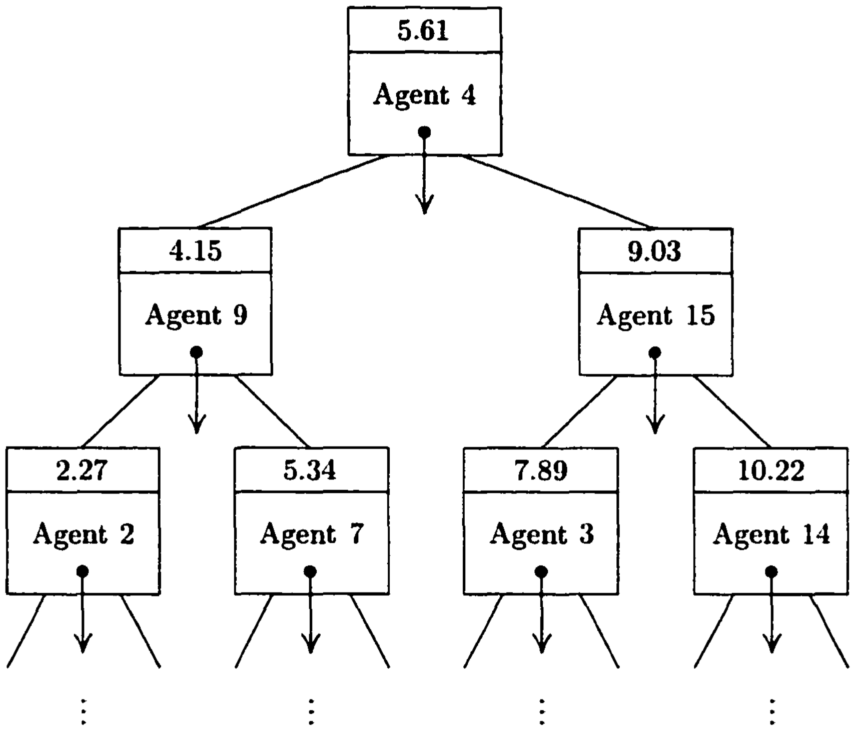


Figure 4.3: A sample splay tree event list

1. If r is the root, the splay terminates.

2. If r has parent q but no grandparent, the required rotation is shown in Figure 4.4.

3. If r has parent q and grandparent p then one of the following four rotations is possible.

   - If q is the right (R) child of p and r is the right (R) child of q, the required RR rotation is shown in Figure 4.5.

   - If q is the left (L) child of p and r is the left (L) child of q, the required LL rotation is symmetric to the RR rotation.

   - If q is the right (R) child of p and r is the left (L) child of q, the required RL rotation is shown in Figure 4.6.

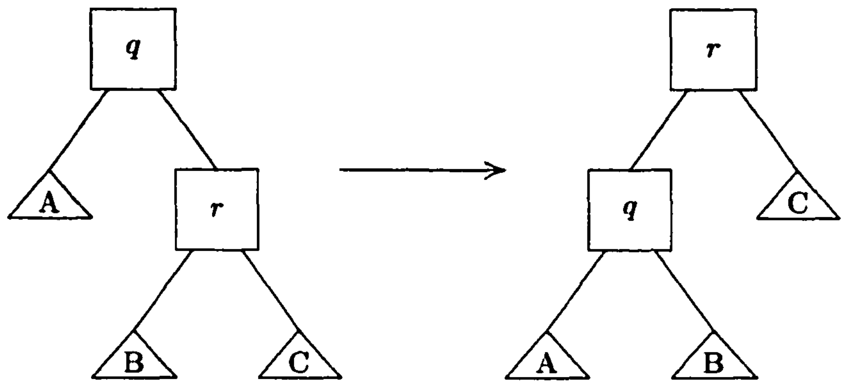


Figure 4.4: Rotation for node r with parent q but no grandparent

- If q is the left (L) child of p and r is the right (R) child of q, the required LR rotation is symmetric to the RL rotation.

Using a splay tree as defined by Sleator and Tarjan, the amortized time complexity of a dequeue operation is $O(\log A(t))$ [98, 100]. Note that single worst-case operations can take linear time; we are guaranteed not that *each* operation is efficient, but that the average cost of all operations is efficient. Similarly, the amortized time complexities of the enqueue and cancel operations are $O(\log A(t))$. Using a splay tree as defined in [100], the time complexities of the three event list operations are summarized in the following table.

Operation	**Complexity**
dequeue	$O(\log A(t))$ (amortized)
enqueue	$O(\log A(t))$ (amortized)
cancel	$O(\log A(t))$ (amortized)

Table 4.3: Time complexities for Sleator-Tarjan splay tree

Jones improved the splay tree implementation specifically for use in discrete-event simulation by eliminating redundant operations [56]. Jones noted that for a dequeue operation,

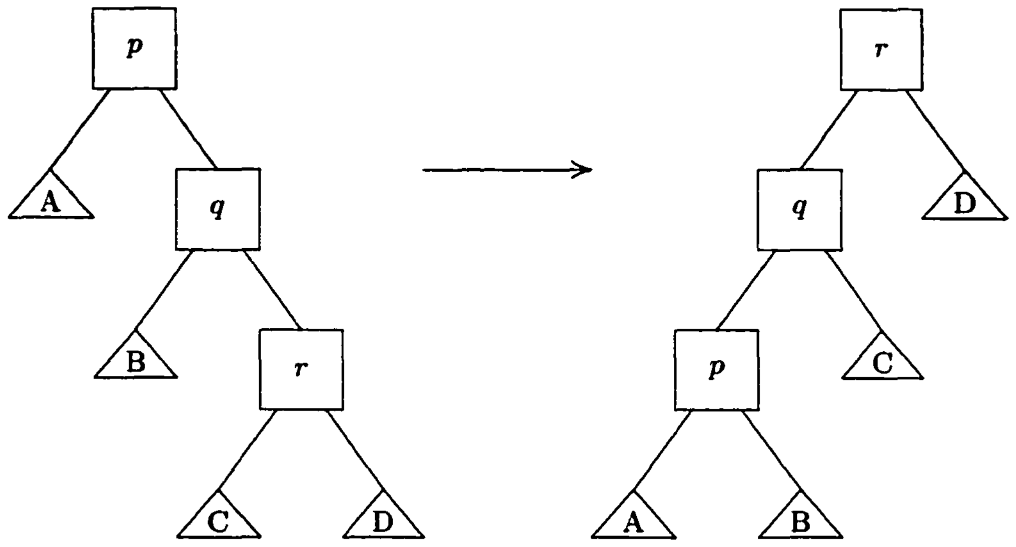


Figure 4.5: RR rotation for splay operation

no priority comparisons are required because the minimum time element will always be the leftmost item in the tree. As a result, splay rotations used to bring the minimum time element to the root of the tree can be eliminated. By keeping track of the location of the current minimum time element in the tree, the dequeue operation can be performed in constant time [67]. Because the Jones-modified splay tree is specialized for the enqueue and dequeue operations of discrete-event simulation, we use its implementation[6] rather than the Sleator-Tarjan splay tree. For a splay tree implemented with Jones's modifications, the time complexities of the three event list operations are summarized in the following table.

Operation	Complexity
dequeue	$O(1)$
enqueue	$O(\log A(t))$ (amortized)
cancel	$O(\log A(t))$ (amortized)

Table 4.4: Time complexities for Jones-modified splay tree

[6]The splay tree implementation is a conversion of Jones's Pascal code referenced in [56].

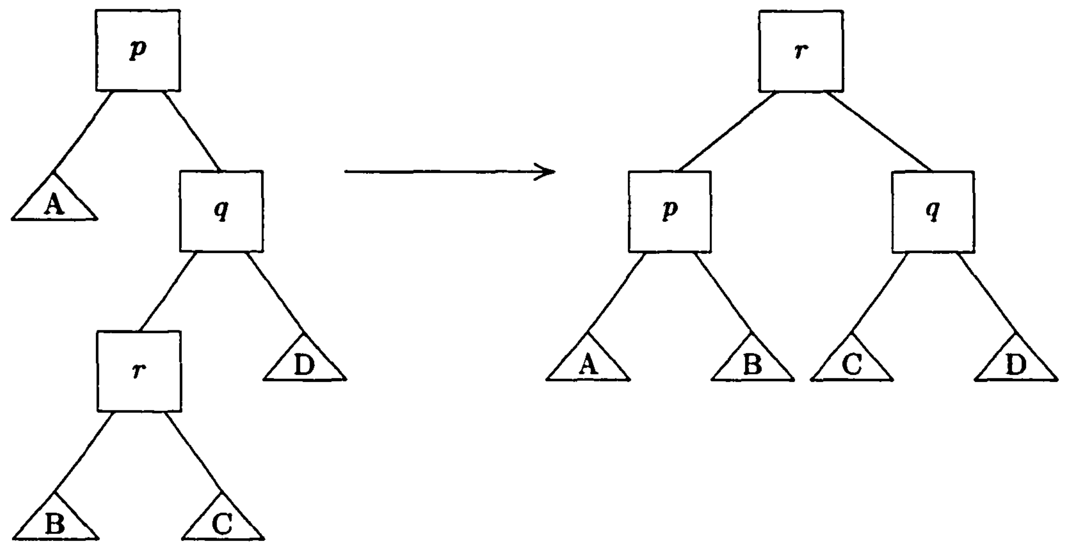


Figure 4.6: RL rotation for splay operation

4.2.4 Calendar Queue

The calendar queue [10] is a multiple-list priority queue used for representing the event list. The calendar queue is modeled after a typical desk calendar. One schedules a future event on the calendar by writing the event in the calendar block corresponding to the day the event will occur. Certain days (blocks) may have multiple events while other days may have no events. Multiple events within the same day are naturally sorted by increasing time.

Upon implementation, each "day" is represented by a sorted linked list, where each element in the list corresponds to an event to occur during that day. An array of pointers, one pointer to each day (i.e., to each linked list), represents the calendar "year". By making the calendar circular, events can be scheduled for up to one year in advance. In addition, events can be scheduled for more than one year in advance by including in each calendar entry the year that the associated event is to occur. Because the calendar is circular, events can be scheduled for *any* year in the future, avoiding the need of an overflow list found in other multiple-list priority queue implementations (e.g., the Lazy Queue [92]).

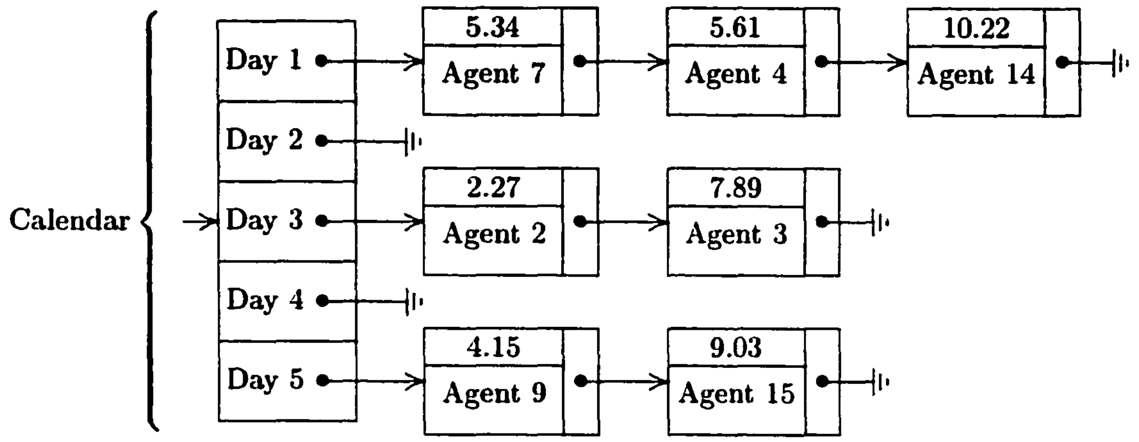


Figure 4.7: A sample calendar queue event list

An example representation of a calendar queue event list for the artificial society model is shown in Figure 4.7. In this example, the calendar consists of five days, each of length 1.00 time units, so that the length of one calendar year is 5.00 time units. Day one of the current year contained events with time stamps in the range [0.00, 1.00); day two contained events with time stamps in the range [1.00, 2.00); and so forth. The third day is the current day of the current year. Moreover, the events to occur within one calendar year of the current day have time stamps 2.27, 4.15, 5.34, and 5.61. The remaining three events in the queue fall outside the length of one year from the current day and so occur during the following year.

For a dequeue operation, the next event to occur is determined by scanning the calendar, starting with the current day, until the first encountered event within the current year is located. If no events occur within the current year, the year is incremented and the process repeated. For an enqueue or cancel operation, the day (i.e., array element) containing the event is determined using a simple hashing function[7], and a linear search of the linked list representing that day determines the correct location of the event.

The length of a year is chosen so that most events (about 75%) occur within one year, i.e., an event is scheduled for no more than one year after the time at which it was enqueued [10]. To avoid long linear searches, the length of a day is chosen so that no day contains an excessive number of events; this length should be roughly the average difference in the time stamps of successive events [10]. In addition, the length of a year and the length of a day are allowed to change dynamically as the number of enqueued events grows and shrinks.

Whenever the number of events exceeds twice the number of days per year, the calendar is copied onto a larger calendar containing twice the number of days as the previous calendar. Similarly, if the number of days per year falls below one-half the number of days per year, the calendar is copied onto a smaller calendar containing half the number of days[8]. The length of a day must also be adjusted accordingly (i.e., either halved or doubled) when copying events onto a new calendar.

Brown argued via intuition and experimentation that the calendar queue has $O(1)$ average performance per operation [10]. However, copying the calendar, as described above, has $O(A(t))$ cost. In addition, Rönngren et al. showed the worst-case amortized cost for an insertion in the calendar queue to be $O(A(t))$ [92].

[7]Let t_e be the time stamp of an event e, d be the length of one day, and n be the number of days in one year. Then the day containing event e is determined using the equation $1 + (t_e/d) \bmod n$.

[8]In this way, the number of days per year is always a power of two.

4.2.5 Spatially Based Multilists

A spatially based multilist is a novel approach for implementing the event list for the artificial society discrete-event simulation model. The multilist is constructed using multiple instances of one of the previous event list implementations, either Henriksen's, splay tree, or calendar queue. Henceforth, we refer to such an instance by the term sublist. The number of sublists S is a compile-time parameter. A container array encapsulates the collection of sublists. Events are distributed among the sublists as described below. A schematic of a multilist with three Henriksen's sublists is depicted in Figure 4.8. The schematic is the same for splay tree or calendar queue sublists except for appropriate sublist modifications.

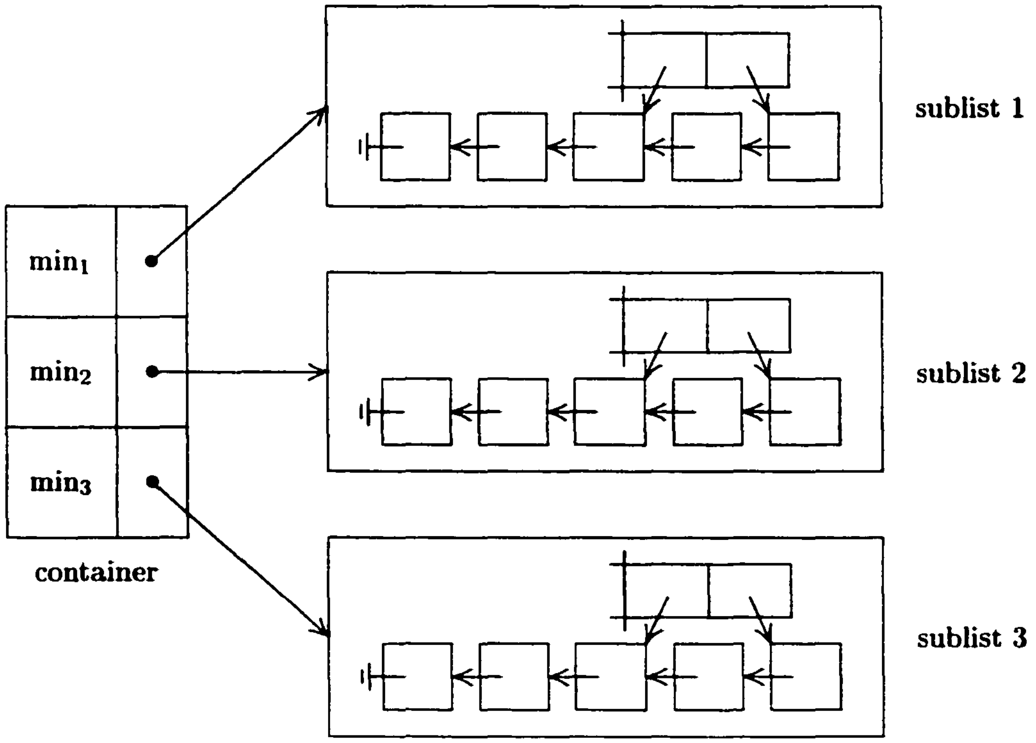


Figure 4.8: A sample multilist event list schematic using Henriksen's sublists

As time evolves, the next event to occur in simulated time resides in exactly one of the sublists. Because this sublist can be *any* of the existing sublists, the time of the event with minimum time stamp in each sublist is maintained in the container array. Admittedly, the overhead of maintaining the sublist minima in the container array is likely to be prohibitive for small models, i.e., for a small number of agents. However, the goal of the spatially based multilist implementation is to judiciously separate the event list into several smaller event lists, thereby reducing the size of any one list to be searched. We expect this separation will be beneficial for very large models.

Newly scheduled events are distributed among the sublists based on the current spatial location on the landscape of the agent associated with that event[9]. Given a predefined number S of sublists, the event for an agent currently located at cell (x, y) is assigned to sublist s via the equation

$$s = \begin{cases} 1 + \frac{\vartheta}{2}, & \vartheta \bmod 2 = 0 \\ 1 + \frac{\vartheta}{2} + \left\lceil \frac{S}{2} \right\rceil, & \vartheta \bmod 2 = 1 \end{cases} \tag{4.1}$$

where $0 \le s < S$ and $\vartheta = ((x \cdot X) + y) \bmod S$.

A potentially frequent spatial bias exists if we assign events to sublists based on contiguous landscape cells. That is, when agents cluster in distinct regions of the landscape, certain sublists can become more heavily loaded than others. Equation 4.1 above attempts to avoid such a bias by assigning the events for agents in contiguous landscape cells to different sublists. As an example, if $S = 3$ the sublist assignments of those events for agents (if any) located in the first eight contiguous landscape cells are shown in Table 4.5. Table 4.6 shows the assignments if $S = 4$.

[9]We also experimented with assigning events to sublists at random. Because the spatially based approach gave smaller execution times than the random approach, we present only the spatially based assignments here.

(x,y)	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	$\cdots$
s	0	2	1	0	2	1	0	2	$\cdots$

Table 4.5: Spatially based multilist sublist assignments with $S = 3$

(x,y)	(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)	(0,6)	(0,7)	$\cdots$
s	0	2	1	3	0	2	1	3	$\cdots$

Table 4.6: Spatially based multilist sublist assignments with $S = 4$

For a dequeue operation, the most imminent event in simulated time is retrieved as follows. Let s be the index of the minimum time stored in the container array. Then the minimum time stamp event from sublist s (i.e., the most imminent event in simulated time) is dequeued according to the protocol of the sublist implementation. The time in position s of the container array is then updated with the time of the *new* minimum time stamp event in sublist s.

For an enqueue operation, the event is assigned to sublist s using Equation 4.1 and the (x, y) position of the associated agent. The event is then enqueued in sublist s according to the protocol of the sublist implementation. If the inserted event is the minimum time stamp event in sublist s, the time in position s of the container array is updated accordingly.

For a cancel operation, the sublist s containing the event to be canceled is determined using Equation 4.1 and the (x, y) position of the associated agent. The event is then located and removed from sublist s according to the protocol of the sublist implementation. If the canceled event was the minimum time stamp event in sublist s, the time in position s of the container array is updated with the time of the *new* minimum time stamp event in sublist s.

4.3 Event List Performance

Although asynchronous time evolution can reduce simulation artifacts in the output produced by an artificial society model, execution time of the simulation can be prohibitively large without a judicious choice for the event list implementation. Therefore we now present performance results for each of the previous event list implementations within the artificial society simulation showing that acceptable execution times can be achieved. Most comparative studies of event list implementations measure execution performance in terms of a single hold operation [56, 67, 91]; however, our interest is in the overall performance for an entire simulation run. Accordingly, the performance measure of interest here is the execution time of the entire simulation.

For all results to be presented, we employ asynchronous time evolution with agent movement and reproduction coupled and gestation period $\eta = 0.0$. We consider the execution time of the simulation using an $X \times Y$ two-peak landscape with $X = Y$ and $X \in \{100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\}$[10]. To evaluate the event list implementations under heavy load, we use the initial conditions presented in Section 3.1. The resulting carrying capacity stabilizes to a proportion of 0.8 of the total cells occupied, and accordingly we initially populate $p = 0.8$ of the total cells at random. We simulate for a maximum time of $T = 1000$ because by that time a stable agent carrying capacity has been achieved. All timing results were obtained using a dedicated Pentium III 1 GHz processor with 1GB of RAM.

[10]The effect of landscape size on execution time is negligible. The execution time is directly related to the size of the event list, i.e., to the number of agents. For convenience, we specify the number of agents as a proportion of the landscape size $X \times Y$.

Figure 4.9 depicts sample execution times for the artificial society simulation initialized as described above using a sorted linked list and Henriksen's event list implementations. As shown, even for a relatively modest event list size (approximately 32 000 events on a 200×200 landscape), the sorted linked list gives a prohibitively large execution time. Comparatively, Henriksen's algorithm provides an execution time on the order of 10 minutes, proving the sorted linked list implementation to be inappropriate in this context.

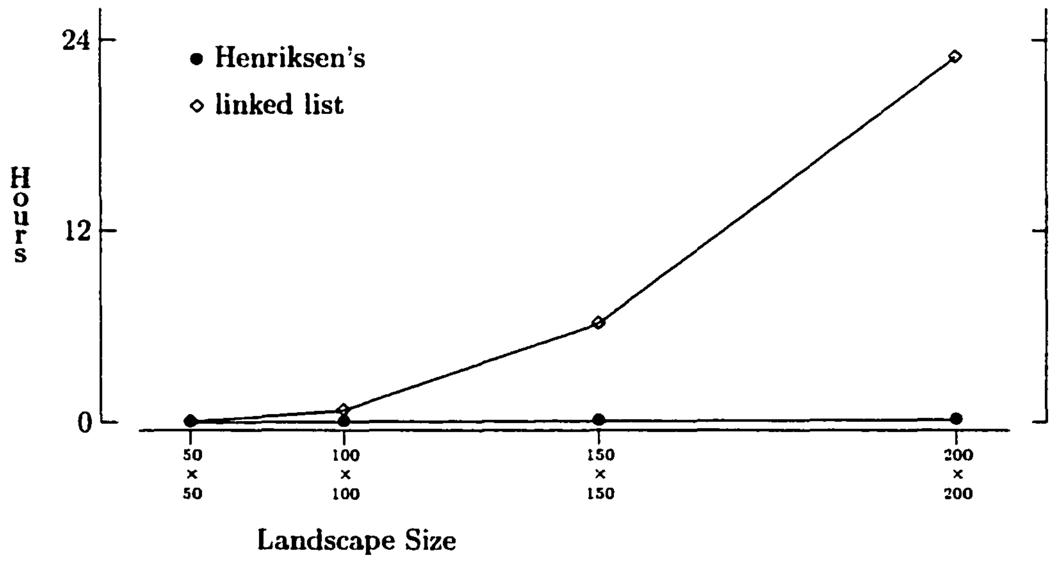


Figure 4.9: Execution times using sorted linked list and Henriksen's event list implementations with $p = 0.8$ of the total cells occupied for increasing landscape sizes

Figure 4.10 depicts sample execution times for the artificial society simulation initialized as described above using Henriksen's, splay tree, and calendar queue event list implementations. Each value displayed is the average time for ten replications along with the corresponding 95% confidence interval (to "fat-dot" accuracy). As shown for the two separate event list loads in Figure 4.10(a) and (b), each of the three event list implementations provides tremendous improvements in execution time compared to the sorted linked list implementation. Note that for a 1000×1000 landscape with heavy load (approximately 800 000

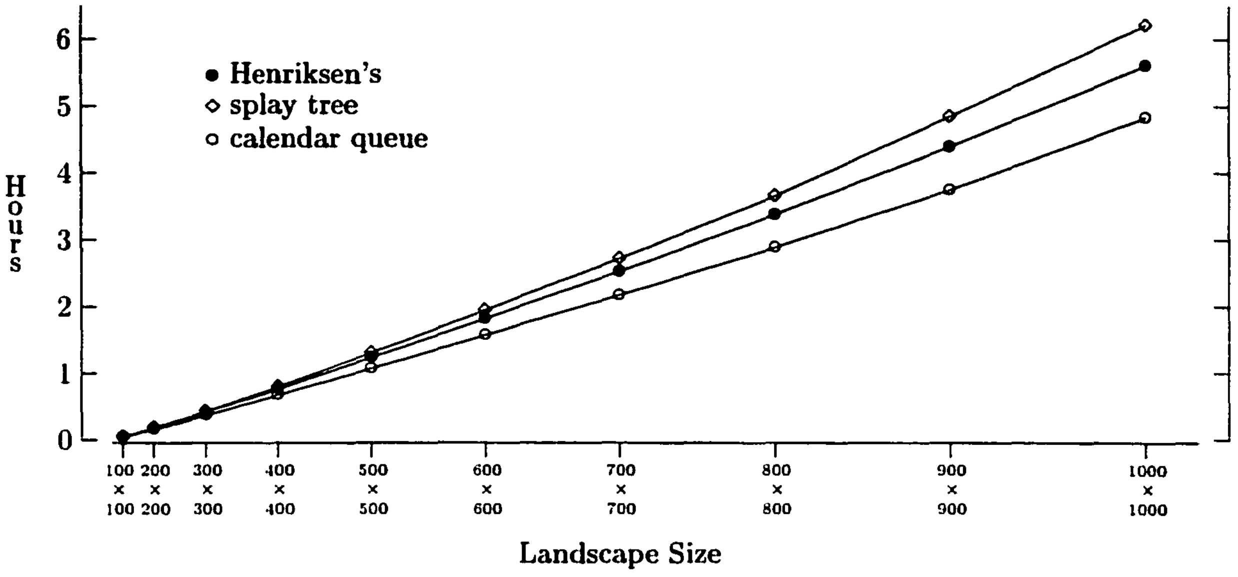


Figure 4.10: Execution times using Henriksen's, splay tree, and calendar queue event list implementations with a proportion $p = 0.8$ of the total cells occupied for increasing landscape sizes

events), all three implementations provide execution times at most one-fourth that of the sorted linked list implementation for a 200 × 200 landscape (approximately 32 000 events). Moreover, of the three implementations, calendar queue yields the best performance with an execution time improvement of nearly an hour or more compared to Henriksen's and the splay tree for a 1000 × 1000 landscape.

Figure 4.11 depicts sample execution times for the artificial society simulation using the spatially based multilist implementations[11] described in Section 4.2.5. Again, each value displayed is the average time for ten replications along with the corresponding 95% confidence interval (to "fat-dot" accuracy). All three multilist implementations provide execution times comparable to the basic implementations shown in Figure 4.10. Note that the calendar queue, the best of the implementations in Figure 4.10, also performs best as a

[11]We experimented with spatially based multilist implementations with two, four, and eight sublists. Because the performances for each were comparable, we present only the results for spatially based multilists with two sublists.

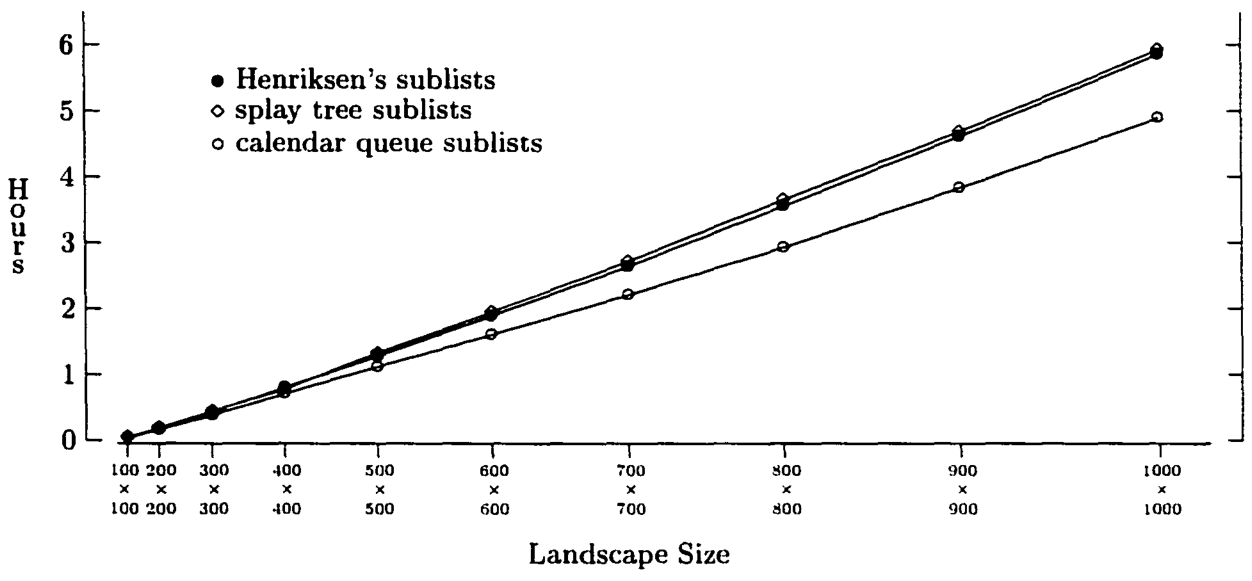


Figure 4.11: Execution times using spatially-based multilist event list implementations with a proportion $p = 0.8$ of the total cells occupied for increasing landscape sizes

sublist implementation.

Furthermore, relative to the results in Figure 4.10, the multilist implementation using Henriksen's sublists does not perform as well as the basic Henriksen's implementation. On a 1000 × 1000 landscape with heavy load, the multilist implementation yields an execution time of nearly one half hour more. Conversely, the multilist implementation using splay tree sublists performs better than the basic splay tree implementation. In this case, on a 1000 × 1000 landscape with heavy load, the multilist implementation yields an execution time of approximately one half hour less. Also note that the multilist implementation using calendar queue sublists performs equivalently to the basic calendar queue implementation.

4.4 Event List Summary

As shown by the results in the previous section, unless the event list in our artificial society model is implemented using an appropriate data structure, asynchronous time evolution leads to prohibitive performance for large models. Using a representative implementation from each of the three basic event list structure types, performance improves dramatically. Moreover, although our spatially based multilist implementations perform comparably to the basic implementations, the multilist only proves beneficial relative to the basic splay tree implementation. In terms of execution time alone, based on these results the calendar queue is the implementation of choice. If simplicity of the implementation is paramount, Henriksen's implementation is much simpler than the calendar queue but provides execution times only slightly worse. In conclusion, the results presented herein show that, as desired, asynchronous time evolution as promoted in Chapter 3 can be incorporated into the artificial society simulation model while maintaining acceptable computational performance.

Chapter 5

Job Scheduling in Parallel Systems

In Chapter 3, we showed that converting the artificial society model from a synchronous to an asynchronous model provides a more realistic model and reduces simulation artifacts. Similarly, in this chapter we show that converting from a synchronous to an asynchronous model improves performance in job scheduling within parallel systems. Our work is based on the idea of *backfilling*, a non-FCFS scheduling policy which permits a select few jobs to jump ahead of a job that cannot begin service immediately. Unlike standard backfilling paradigms which use a single queue of jobs, we transform the system into an asynchronous model by splitting the system into multiple disjoint partitions, with one queue per partition. Jobs are classified and assigned to one of these partitions according to the estimated job duration. We show that such classification improves the average job slowdown by reducing the likelihood that a short job is overly delayed in the queue behind a very long job.

In this chapter, we present a detailed description of, and results from, our new scheduling policy. We begin with a discussion of background information and an analysis of the workloads used in our study, then describe our new backfilling policy in detail, and conclude by presenting performance comparisons of our new policy in relation to the standard backfilling policy.

policies.

Backfilling has been proposed as a more efficient alternative to simple static FCFS schedulers [78, 102]. Backfilling reduces resource fragmentation and increases system utilization by executing jobs in an order different than their arrival order, thereby exploiting otherwise unused processors. Users are expected to provide nearly accurate estimates of the job execution times. Using these estimates, the scheduler rearranges the waiting queue, allowing short jobs to move ahead of long jobs provided certain previously submitted jobs are not delayed. Various versions of backfilling have been proposed [58, 78, 86]. Keleher et al. characterize the effect of job length and parallelism on backfilling performance [58]. Perkovic and Keleher propose sorting by job length to improve backfilling and introduce the idea of speculative execution, in which long jobs are given a short trial execution to detect whether or not the jobs crash [86].

Industrial schedulers that are widely accepted by the high performance community, including the Maui Scheduler [8, 69], PBS Scheduler [8, 85], and IBM LoadLeveler [52], offer a variety of configuration parameters. In these schedulers, available configuration parameters include multiple queues to which different job classes are assigned, multiple job priorities, multiple scheduling policies per queue, and the ability to treat interactive jobs differently from batch jobs. The immediate benefit of such flexibility in policy parameterization is the ability to customize the scheduling policy according to the site's needs. However, optimal policy customization to meet the needs of an ever changing workload is an elusive goal.

In this work, we propose a simple yet effective batch scheduler policy that is based on the extensively analyzed *aggressive* backfilling strategy [78]. Our policy is inspired by related work in task assignment for distributed servers that strongly encourages separation of jobs according to their length, especially for workloads with execution times characterized by long-tailed distributions [90, 97]. Similarly, observed high variance in job execution times in parallel workload traces advocates separating long from short jobs in parallel schedulers. In contrast to other backfilling related works, our policy maintains multiple queues and effectively separates short from long jobs. Our policy requires only an *a priori* definition of job classes and then the policy adjusts its processor-to-class allocations automatically. Furthermore, we employ speculative execution to ensure better job separation according to the actual job execution time. The effective separation of jobs coupled with speculative execution permits our policy to outperform the standard backfilling policy in variety of contexts. In the following section, we present detailed analyses of the workload traces used in the simulation to evaluate our proposed policy.

5.2 Scheduling Workload Analysis

The difficulty of scheduling parallel resources is deeply interwoven with the inherent variability in parallel workloads. Because our goal is to propose a robust policy that works efficiently regardless of the workload type, we first closely examine real parallel workloads of production systems. We select four workload logs from the parallel workload archive [30]. Each log provides the arrival time of each job (i.e., the job submit time), the number of processors requested, the estimated service time of the job, the actual service time of the job, the start time of the job, and possible additional resource requests (e.g., memory per node). The selected traces are summarized below.

- **CTC**: This trace contains entries for 79 302 jobs that were executed on a 512-node IBM SP2 at the Cornell Theory Center from July 1996 through May 1997.
- **KTH** : This trace contains entries for 28 487 jobs executed on a 100-node IBM SP2 at the Swedish Royal Institute of Technology from October 1996 through August 1997.
- **PAR**: This trace contains entries for 37 910 jobs that were executed on a 416-node Intel Paragon at the San Diego Supercomputer Center during 1996. Because this trace contains no user estimates, we use the actual run times as accurate estimates.
- **SP2**: This trace contains entries for 67 665 jobs that were executed on a 128-node IBM SP2 at the San Diego Supercomputing Center from May 1998 through April 2000.

5.2.1 The Arrival Process

The arrival process significantly affects the performance and scheduling decisions in any queueing system. To visualize the time evolution of the arrival process, in Figure 5.1 we plot for each trace the total number of arriving jobs per week as a function of time. We observe bursts in the arrival process[1], but not of the same magnitude as the "flash crowds" experienced by web servers [90]. Significant differences in the per-week arrival intensity exist within each workload, as well as across all workloads. For this reason we focus not only on *aggregate* statistics (i.e., the average performance measures obtained after simulating the system using the entire workload trace), but also on *transient* statistics within specific time windows.

[1]Bursts also exist relative to smaller time units (e.g., days and hours), but such graphs are omitted for the sake of brevity.

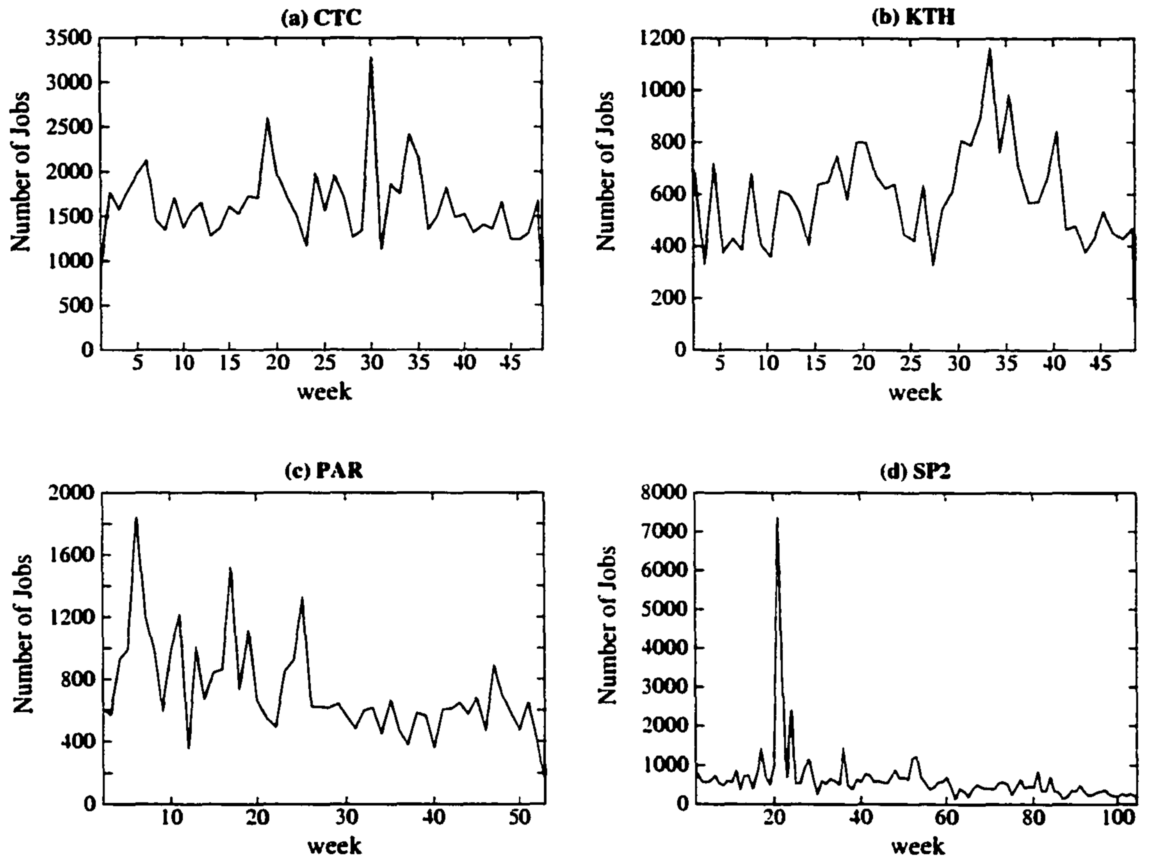


Figure 5.1: Total number of arriving jobs per week as a function of time (weeks)

5.2.2 The Service Process

In addition to the arrival process, the service process also affects scheduling decisions and performance in a queueing system. Table 5.1 provides summary statistics for the selected traces[2]. Observe the wide disparity of the mean *actual* job service time across workloads. Also notice the difference (of as much as two orders of magnitude) between the mean and the median actual service times within a workload. The high coefficients of variation (C.V.)

[2]A common characteristic in many of these traces is that the system administrator places an upper limit on the job service time. If this limit is reached, the job is killed. Our statistics include the terminated jobs; therefore, some of our output statistics are higher than those reported elsewhere (e.g., see [29]).

in actual job service times coupled with the large differences between mean and median values suggest the existence of a "fat tail" in the distribution of service times. Log-log complementary distribution plots confirm the absence of a heavy tail in the distributions [29], but actual run times nonetheless remain very skewed within each workload. This type of distribution advocates separating jobs according to their duration to different queues in order to minimize queueing time of short jobs that are delayed behind very long jobs.

Workload	Mean Actual Service Time	Median Actual Service Time	CV Actual Service Time	Mean Estim. Service Time	Median Estim. Service Time	CV Estim. Service Time
CTC	10 983.42	946	1.65	24 323.75	10 800	1.07
KTH	8 877.07	847	2.34	13 678.01	4 200	1.80
PAR	7 000.02	155	1.90	—	—	—
SP2	6 118.96	514	2.37	14 337.18	5 400	1.39

Table 5.1: Summary statistics of the four selected workloads. All times are reported in seconds.

The differences between the mean and median values for *estimated* service times are also large (of nearly an order of magnitude difference), and the C.V. of estimated service times are also large. More importantly, however, are the discrepancies between the actual and estimated service times. Notice that the mean estimated service time is consistently twice the size of the mean actual service time. Also note that the differences between the median actual and median estimated service times are an order of magnitude difference or more. These differences are evidence that users tend to overestimate the service times of jobs to prevent jobs from being killed. As a result, to develop a robust scheduling policy we must evaluate the classification policy that separates jobs into different queues within the context of accurate estimates (i.e., using actual service times from the workloads as estimates) and inaccurate estimates (i.e., using user-provided service estimates from the workloads).

5.2.2.1 Classification Using Accurate Service Estimates

We first consider the classification policy assuming accurate service estimates. We use the actual job service times as accurate estimates and, because Table 5.1 indicates wide variation in actual job service times, we classify jobs according to these estimates. After experimenting with several different classifications, we select the following four-part classification. Although no classification will give an *equal* proportion of jobs per class across all workloads, our classification provides a representative proportion of jobs per class, as shown in Figure 5.2.

- **class 1**: Short jobs have actual service time $\leq$ 100 seconds.
- **class 2**: Medium jobs have actual service time $>$ 100 seconds and $\leq$ 1000 seconds.
- **class 3**: Long jobs have actual service time $>$ 1000 seconds and $\leq$ 10 000 seconds.
- **class 4**: Extra-long jobs have actual service time $>$ 10 000 seconds.

Figure 5.2 presents the actual service time characteristics of the four selected workloads. The left column depicts the overall and per-class mean actual job service time as a function of the trace time[3]. The center column depicts the overall and per-class C.V. of the actual job service time. Finally, the right column depicts the proportion of jobs per class.

As shown in Figure 5.2, the mean actual job service times and the overall C.V. (solid line) vary significantly across time. As expected, for all workloads the per-class C.V. is considerably smaller than the overall C.V. For all four traces the proportion of jobs in each

[3]For statistical significance, we compute workload statistics for batches of 1000 jobs, but plot each batch as a function of the arrival time of the first job in the batch.

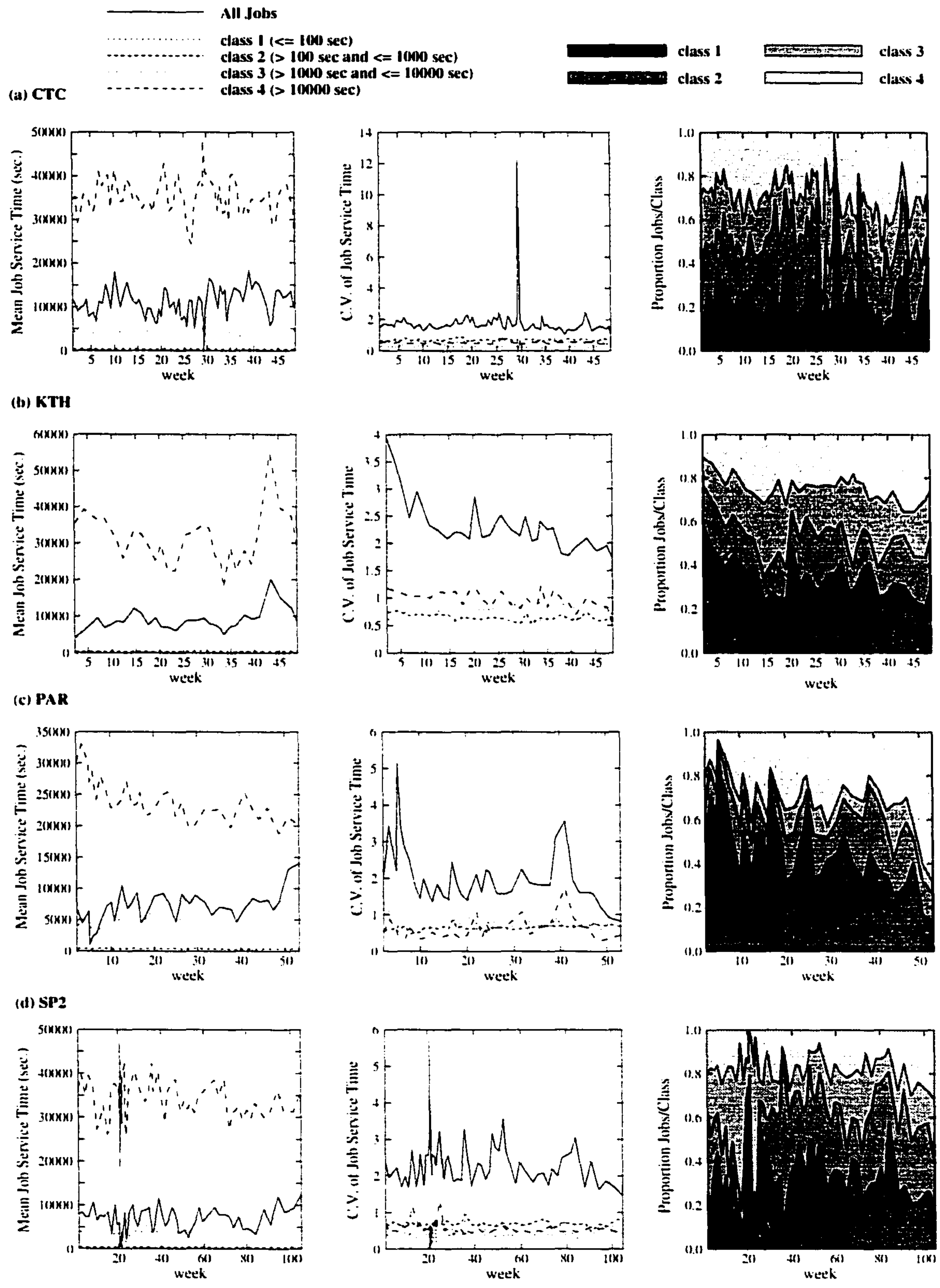


Figure 5.2: Actual service time characteristics of the four workloads using four-part classification

class varies dramatically with time. In other words, duration demands are very different from week to week. Considerable variation is also present in the proportion of jobs per class. For example, the PAR workload consists primarily of short, i.e., class one, jobs. For all four workloads, the proportion of jobs in each class varies dramatically from week to week.

5.2.2.2 Classification Using Inaccurate Service Estimates

We now consider the classification policy assuming inaccurate estimates. Figure 5.3 depicts the average estimated and average actual service times versus time for the three traces that provide estimates. As shown, user estimates are consistently twice the actual service time. While the four-part classification in the previous section works well for accurate estimates, inaccurate estimates lead to inappropriate classification of many jobs. That is, overestimation can lead to a job that, based on its actual service time, *should* be grouped into a class different than the class selected based on the estimate. As such, short jobs may be unwittingly grouped with long jobs, thereby defeating the goal of separating jobs based on job duration. In addition, because of the overestimates very few jobs will be classified as short jobs, effectively rendering the first class useless.

To address these problems, we assume that a user will naturally overestimate the service time of a job to prevent the job from being killed. In effect, we collapse the four-part classification into the following three-part classification based on workload analysis that shows large overestimates.

- **class 1**: Short jobs have estimated service time $\leq$ 1000 seconds.

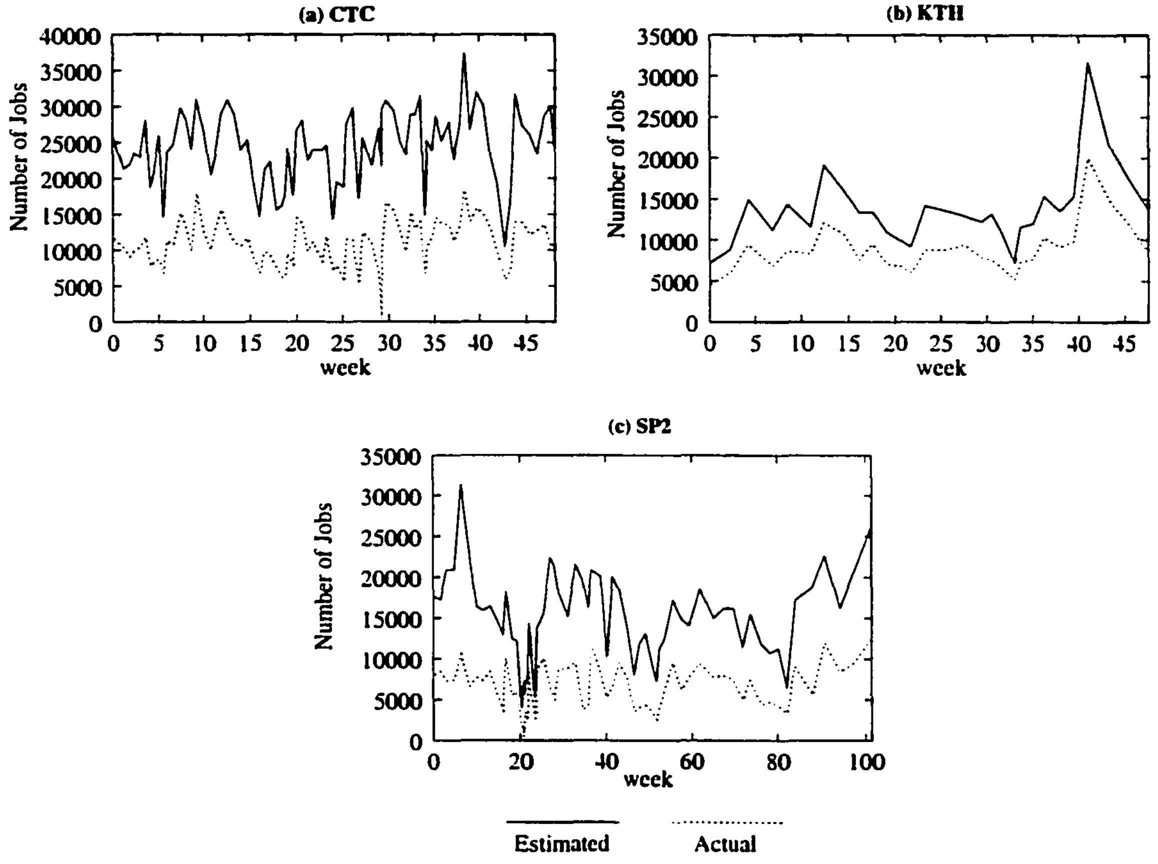


Figure 5.3: Service time overestimates as a function of time

- **class 2**: Medium jobs have estimated service time > 1000 seconds and ≤ 10 000 seconds.
- **class 3**: Long jobs have estimated service time > 10 000 seconds.

As we will show, this three-part classification coupled with speculative execution permits our proposed backfilling policy to perform well even in the presence of poor user estimates.

In summary, we see that workloads vary significantly and follow no discernible trends as they evolve in time. In addition, the classification of jobs is very dependent on whether or not user estimates of service times are accurate. Despite such variability, in the following section we describe a novel, robust scheduling policy that offers performance guarantees under such transient workload conditions.

5.3 Scheduling Policies

In this section, we describe a new job scheduling policy, based on backfilling, that adapts its own parameters according to changing workload conditions. Our policy divides the system into multiple partitions, one queue per partition, to effectively separate jobs according to their duration. The goal is to reduce the average job slowdown by decreasing the number of short jobs delayed in the queue behind long jobs. Before introducing our new policy, we first describe the basic single-queue backfilling paradigm.

5.3.1 Single-Queue Backfilling

Backfilling is a commonly used scheduling policy that attempts to minimize fragmentation of system resources by executing jobs in an order different than their submission order [58, 78]. A job that is backfilled is allowed to jump ahead of jobs that arrived earlier (but are delayed because of insufficient idle processors) in an attempt to exploit otherwise currently idle processors. The order of job execution is handled differently by two types of backfilling. *Conservative* backfilling permits a job to be backfilled provided it does not delay *any* previous job in the queue. *Aggressive* backfilling ensures only that the *first* job in the queue is not delayed. We consider only aggressive backfilling because results have shown its performance to be superior to conservative backfilling [78].

Basic aggressive backfilling is a non-preemptive, space-sharing policy that assumes a single queue of jobs to be executed. Jobs enter this queue when submitted by the user.

Each job is characterized by its arrival time, by the number of processors required (i.e., the job width), and by an estimate of the expected service time. Any job that attempts to execute for a time greater than its estimated service time is terminated by the system.

The single-queue backfilling policy always attempts to backfill as many queued jobs as possible. In general, the process of backfilling *exactly one* of these many jobs occurs as follows. Define the *pivot job* to be the first job in the queue. If there are currently idle processors sufficient for the pivot job, the scheduler starts executing the pivot immediately, and a new pivot is defined appropriately. Otherwise, the scheduler sorts all currently executing jobs in order of their expected completion time. The scheduler can then determine the *pivot time*, i.e., the time when sufficient processors will be available for the pivot job. At the pivot time, any idle processors not required for the pivot job are denoted as *extra* processors. The scheduler then searches for the first queued job that

- requires no more than the number of currently idle processors *and* will finish by the pivot time, or
- requires no more than min{currently idle processors, extra processors}.

If such a job is found, the job is backfilled, i.e., the scheduler starts executing the job immediately; otherwise, the scheduler continues searching the list of queued jobs until either a job is backfilled or the search is exhausted.

This process of backfilling exactly one job is repeated until all queued jobs have been considered for backfilling. Hence, the single-queue backfilling policy attempts to backfill as many jobs as possible until no more jobs can be backfilled. This basic single-queue aggressive backfilling algorithm, employed whenever a job is submitted to the system or whenever a job completes execution, is outlined in Algorithm 5.1.

```
for (all jobs in queue) {
  pivot = first job in queue;
  pivot time = time when sufficient processors will be available for pivot job;
  extra = idle processors at pivot time not required by pivot;
  if (job is pivot) {
    if (current time equals pivot time)
      start job immediately;
  } else {
    if (job requires ≤ currently idle procs and will finish by pivot time)
      start job immediately;
    else if (job requires ≤ min{currently idle procs, extra procs})
      start job immediately;
  }
}
```

Algorithm 5.1: Basic single-queue aggressive backfilling algorithm

Single-queue aggressive backfilling ensures that once a job becomes the pivot, it cannot be delayed. A job may be delayed in the queue before becoming the pivot, but when the job reaches the front of the queue, the job is assigned a scheduled starting time. If a currently executing job finishes early, the pivot may begin executing earlier than its assigned starting time, but it will *never* begin executing after the assigned starting time.

5.3.2 Multiple-Queue Backfilling Using Accurate Estimates

Because the performance of any scheduling policy is sensitive to the transient nature of the impending workload, we propose a multiple-queue backfilling policy that permits the scheduler to quickly change parameters in response to workload fluctuations. Our goal is to decrease the average job slowdown by reducing the number of short jobs delayed in the queue behind longer jobs.

The multiple-queue backfilling policy splits the system into multiple *disjoint* partitions, with each partition assigned its own separate queue of jobs. The splitting is accomplished by classifying jobs and assigning them to one of the partitions according to job duration as described in Section 5.2.2.1. As jobs are submitted to the system, they are placed in the queue in exactly one of these partitions based on the user estimate of service time. Let t_e be the estimate (in seconds) of the service time of a submitted job. In the presence of accurate estimates, we advocate the use of four separate queues, one per job class (i.e., per system partition), and assign the job to the queue in partition p according to the following equation, consistent with the job classification presented in Section 5.2.2.1.

$$p = \begin{cases} 1, & 0 < t_e \leq 100 \\ 2, & 100 < t_e \leq 1000 \\ 3, & 1000 < t_e \leq 10\,000 \\ 4, & 10\,000 < t_e \end{cases} \qquad \text{for accurate } t_e$$

Note that the assignment of a job to a queue is based solely on the user estimate of job service time and *not* on the number of requested processors. Initially, the processors are distributed evenly among the partitions. As time evolves, processors may move from one partition to another (i.e., the partitions may contract or expand) so that currently idle processors in one partition can be used for immediate backfilling in another partition. Hence, as shown in Figure 5.4 the partition boundaries become dynamic, allowing the system to adapt itself to changing workload conditions. We stress that the policy does not starve a job that requires the entire machine for execution. When such a job is ready to begin executing (according to the job arrival order), the scheduler allocates all processors to the partition where the job is assigned. After the job completes, the processors will be redistributed among the partitions according to the ongoing processor demands of each partition.

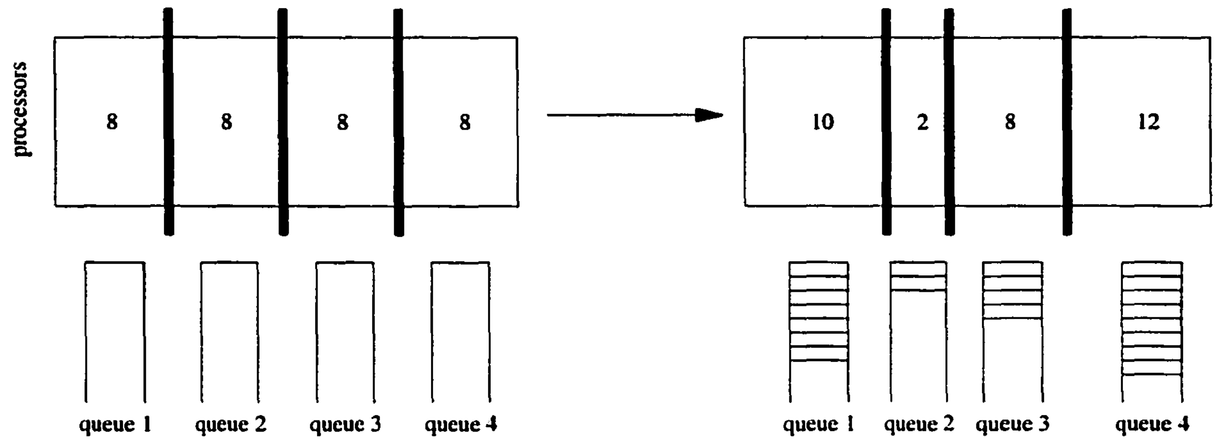


Figure 5.4: An example 32-processor system in which multiple-queue backfilling permits the four initial partition boundaries to adapt as workload conditions change

The multiple-queue backfilling policy considers all queued jobs (one at a time, in the order of arrival regardless of queue). Similar to the single-queue backfilling policy, define the following:

- idle_p: the number of currently idle processors in partition p;
- pivot_p: the first job in the queue in partition p;
- pivot-time_p: the scheduled starting time for pivot_p (i.e., the earliest time when sufficient processors will be available for pivot_p); and
- extra_p: the number of idle processors in partition p at pivot-time_p not required for pivot_p.

The sufficient processors available at pivot-time_p consist of idle_p and, if necessary, some combination of idle and/or extra processors from other partitions such that no other pivot that arrived earlier than pivot_p is delayed. The assignment of a scheduled starting time to a pivot job will never delay any current pivot in another partition (i.e., any other pivot that arrived earlier), suggesting that the algorithm is deadlock-free.

The policy always attempts to backfill as many queued jobs as possible. In general, *exactly one* of these many jobs is backfilled as follows. Let p be the partition to which the job belongs. If the job is pivot$_p$, the scheduler starts executing the job immediately only if the current time is equal to pivot-time$_p$, in which case a new pivot$_p$ is defined appropriately. If the job is not pivot$_p$, the scheduler starts executing the job immediately only if there are sufficient idle processors in partition p without delaying pivot$_p$, or if the partition can take idle processors sufficient to meet the job's requirements from one or more other partitions without delaying any pivot.

This process of backfilling one job is repeated, one job at a time in the order of arrival regardless of queue, until all queued jobs have been considered for backfilling. Hence, the multiple-queue backfilling policy attempts to backfill as many jobs as possible until no more jobs can be backfilled. This multiple-queue aggressive backfilling algorithm, employed whenever a job is submitted to the system or whenever a job completes execution, is outlined in Algorithm 5.2.

5.3.3 Multiple-Queue Backfilling Using Inaccurate Estimates

In the presence of inaccurate estimates, the four-part classification for multiple-queue backfilling suffers because short jobs can be unwittingly grouped with long jobs based on poor user estimates. As such, in this case we advocate the use of three separate queues to account for user overestimates. Let t_e be the estimate (in seconds) of the service time of a submitted job. We assign the job to the queue in partition p according to the following

```
for (all jobs in order of arrival) {
  $p$ = partition to which job is assigned;
  pivot$_p$ = first job in queue in partition $p$;
  pivot-time$_p$ = earliest time when sufficient procs (from this and perhaps other
      partitions) will be available for pivot$_p$;
  extra$_p$ = idle processors in partition $p$ at pivot-time$_p$ not used by pivot$_p$;
  if (job is pivot$_p$) {
    if (current time equals pivot-time$_p$) {
      reassign procs (if required) from other partitions to partition $p$;
      start job immediately;
    }
  } else {
    if (job requires $\leq$ idle$_p$ and will finish by pivot-time$_p$)
      start job immediately;
    else if (job requires $\leq$ min{idle$_p$, extra$_p$})
      start job immediately;
    else if (job requires $\leq$ idle$_p$ plus some combination of idle/extra procs
              from other partitions such that no pivot is delayed) {
      reassign necessary procs from other partitions to partition $p$;
      start job immediately;
    }
  }
}
```

Algorithm 5.2: Multiple-queue aggressive backfilling algorithm.

equation, consistent with the job classification presented in Section 5.2.2.2.

$$p = \begin{cases} 1, & 0 < t_e < 1000 \\ 2, & 1000 \leq t_e < 10\,000 \\ 3, & 10\,000 \leq t_e \end{cases} \qquad \text{for inaccurate } t_e$$

Additionally, with inaccurate estimates the four-part classification suffers because many jobs appear to crash. Table 5.2 shows the significant proportion of total jobs that have estimated run times greater than 1000 seconds but actual run times less than 180 seconds. As a result, many jobs with short actual service times but long estimates are inappropriately grouped with long jobs. Because the impact of queueing delay is much more profound on short jobs than long, a significant decline in the average job slowdown results.

Trace	Total Jobs	Crashed Jobs	Proportion
CTC	79 302	12 903	0.16
KTH	28 487	3000	0.11
SP2	67 665	15 974	0.24

Table 5.2: Proportions of (possibly) crashed jobs for three parallel workload traces

To remedy this situation, we also employ speculative execution of jobs[4] to weed out a large proportion of crashed jobs [86]. If the estimated service time of a submitted job is greater than 1000 seconds (i.e., belongs to partition two or three), the job is immediately scheduled for speculative execution for a maximum of 180 seconds[5]. If the job does not terminate within the allotted 180 seconds, the job is killed and is then placed into the queue in partition p (determined using the estimated service time) in increasing order of arrival time. Then, according to the multiple-queue backfilling policy, the job will be subject to future scheduling for its full estimated service time.

5.3.4 Backfilling with Job Priorities and Reservations

Scheduling jobs on a site that is part of a computational grid imposes additional challenges. The policy must cater to three classes of jobs: local jobs (parallel or sequential) that should be executed in a timely manner, jobs external to the site that do not have high priority (i.e., jobs that can execute when the system is not busy serving local jobs), and external jobs that require reservations (i.e., jobs that require resources within a very restricted time frame to be successful).

Within the context of scheduling resources in a computational grid, we therefore sup-

[4]Within the context of real systems, as a general rule jobs cannot be killed and restarted. Speculative execution can be used, however, by permitting a user to flag a job as restartable (when appropriate) with the anticipation of improved slowdown.

[5]We experimented with speculative execution times from one to five minutes. Speculative execution for a maximum of three minutes removes most of the jobs that appear to crash, as depicted in Table 5.2.

plement the single- and multiple-queue backfilling policies by considering static job priority levels and job reservations.

- We consider jobs submitted by local users to have high priority and those jobs submitted by external sources (i.e., from elsewhere in the computational grid) to have low priority. Our goal is to serve these external jobs as quickly as possible without inflicting delays on local jobs.

- We assume that the system also serves jobs that require reservations. Our goal is to serve these jobs as close to the requested reservation time as possible regardless of the consequences on remaining jobs.

5.3.4.1 Single-Queue Backfilling with Priorities

Single-queue backfilling is modified to incorporate job priorities as follows. If a job being inserted into the queue has high priority, the job is placed into the queue after any queued high priority jobs that arrived before it. If the job has low priority, it is placed into the queue after all queued high priority jobs *and* after any queued low priority jobs that arrived before it. This queue ordering ensures that all high priority jobs are considered for backfilling before any low priority job. Note that a high priority job will immediately become the pivot if the current pivot has low priority; the displaced low priority pivot is placed into the queue in order of arrival. Also note that any job inserted into the queue after speculative execution will replace a pivot job of the same priority that has later arrival time.

5.3.4.2 Multiple-Queue Backfilling with Priorities

Multiple-queue backfilling is modified to incorporate job priorities as follows. If a job is classified and assigned to partition p, the job is placed into the queue in partition p after any queued high priority jobs in partition p that arrived before it. If the job has low priority, it is placed into the queue in partition p after all queued high priority jobs *and* after any queued low priority jobs in partition p that arrived before it. The multiple-queue policy considers high priority jobs first for backfilling (in their order of arrival, regardless of partition) followed by low priority jobs (in their order of arrival, regardless of partition). Similar to the single-queue policy, a high priority job assigned to partition p will immediately become pivot$_p$ if the current pivot$_p$ has low priority; the displaced low priority pivot$_p$ is placed into the queue in partition p as described above. Also note that any job inserted into the queue in partition p after speculative execution will replace pivot$_p$ of the same priority if the latter has greater arrival time.

5.3.4.3 Backfilling with Reservations

Apart from job priorities, we also briefly describe here the inclusion of job reservations in both multiple-queue and single-queue backfilling. A user may schedule a reservation for future execution of a job if, for example, a dedicated environment is desired. Accordingly, when a request for a reservation is submitted, the scheduler determines the earliest time greater than or equal to the requested reservation time when the job can be serviced, and immediately schedules the job for execution at that time. For simplicity, we assume that once a job receives a reservation, the reservation will not be canceled nor can the time of the reservation be changed. Furthermore, we assume that all non-reservation jobs are of the same priority. Therefore, the process of backfilling with reservations in the single-queue and the multiple-queue policies remains as described before, with the exception that backfilling must respect all reservations.

5.3.5 Scheduling Policies Summary

In summary, both the single-queue and multiple-queue policies backfill jobs in order to exploit idle processors and reduce system fragmentation. Both policies ensure that once a job reaches the front of the queue, it cannot be delayed. However, by classifying jobs according to job length, the multiple-queue policy reduces the likelihood that a short job will be overly delayed in the queue behind a very long job. Additionally, because processors are permitted to cross partition boundaries, the multiple-queue policy can quickly adapt to a continuously changing workload. Unlike commercial schedulers that typically are difficult to parameterize, multiple-queue backfilling requires only an *a priori* definition of job classes, and then the policy automatically adjusts the processor-to-class allocations.

5.4 Performance Analysis

In this section, we evaluate via simulation the performance of the single-queue and multiple-queue backfilling policies presented in the previous section. Our simulation experiments are driven using the four workload traces from the Parallel Workload Archive described in Section 5.2. From the traces, for each job we extract the job's arrival time, the user estimate of the job service time (if available), the actual job service time, and the number of requested processors. Because we do not use the job completion times from the traces, the scheduling strategies used on the corresponding systems are not relevant to our study.

Consequently, our experiments fully capture the fluctuations in job arrival rate and service demands.

We consider *aggregate* performance measures, i.e., average statistics computed for all jobs for the entire experiment, and *transient* performance measures, i.e., snapshot statistics for batches of 1000 jobs that are plotted across the experiment time and illustrate how well the policy reacts to sudden changes in the workload. The performance measure of interest here is the job slowdown s defined by the equation

$$s = 1 + \frac{d}{\nu}$$

where d and ν are respectively the average delay time and actual service time of a job[6]. To compare the performance results of multiple-queue backfilling with standard single-queue backfilling, we also define the slowdown ratio $\mathcal{R}$ by the equation

$$\mathcal{R} = \frac{s_1 - s_m}{\min\{s_1, s_m\}}$$

where s_1 and s_m are the single-queue and multiple-queue average slowdowns respectively[7]. $\mathcal{R} > 0$ indicates a performance gain obtained using multiple queues relative to a single queue; $\mathcal{R} < 0$ indicates a performance loss suffered using multiple queues relative to a single queue.

[6]Bounded slowdown [78] is another popular performance measure. Because the performance of the two policies is qualitatively the same using either of the two measures, we omit performance results obtained for bounded slowdown.

[7]Because of the $\min\{s_1, s_m\}$ term in the denominator, $\mathcal{R}$ is a *fair*, properly scaled measure of the performance that equally quantifies gain or loss experienced using multiple queues relative to a single queue. If we instead use s_m (or s_1) in the denominator, we bias the measure toward gains (or losses).

5.4.1 Backfilling Policy Comparison Using Accurate Estimates

We first compare the performance of the two backfilling policies using actual service times from the workloads as accurate estimates. Because the estimates are accurate, for the results in this section we compare the performance of multiple-queue backfilling (using the four-part classification described in Section 5.2.2.1) to single-queue backfilling, both *without* speculative execution.

Figure 5.5 depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling (computed using the average slowdown obtained using each policy) for each of the four traces. Figure 5.5(a) depicts $\mathcal{R}$ for all job classes combined, while Figures 5.5(b)–(e) each depict $\mathcal{R}$ for an individual job class. As shown in Figure 5.5(a), multiple-queue backfilling provides better overall average job slowdown (i.e., $\mathcal{R} > 0$). With the exception of the extra-long job class, multiple-queue backfilling also provides better average job slowdown within each of the individual job classes, as shown in Figures 5.5(b)–(e).

By separating jobs into separate queues, a queued job competes directly only with jobs in the same queue for access to resources. Relative to using a single queue, short jobs therefore tend to gain access to resources more quickly, while long jobs tend to be delayed slightly. As a result, short jobs are assisted at the expense of long jobs using the multiple-queue policy, thereby improving the average job slowdown. Figures 5.5(b)–(e) confirm that, by splitting the system into multiple partitions, we manage to reduce the number of short jobs delayed behind extra-long jobs. Across all workloads, jobs belonging to all but the extra-long job class achieve significant performance gains. Extra-long jobs experience a decline in average slowdown, but the magnitude of decline is generally much less than the

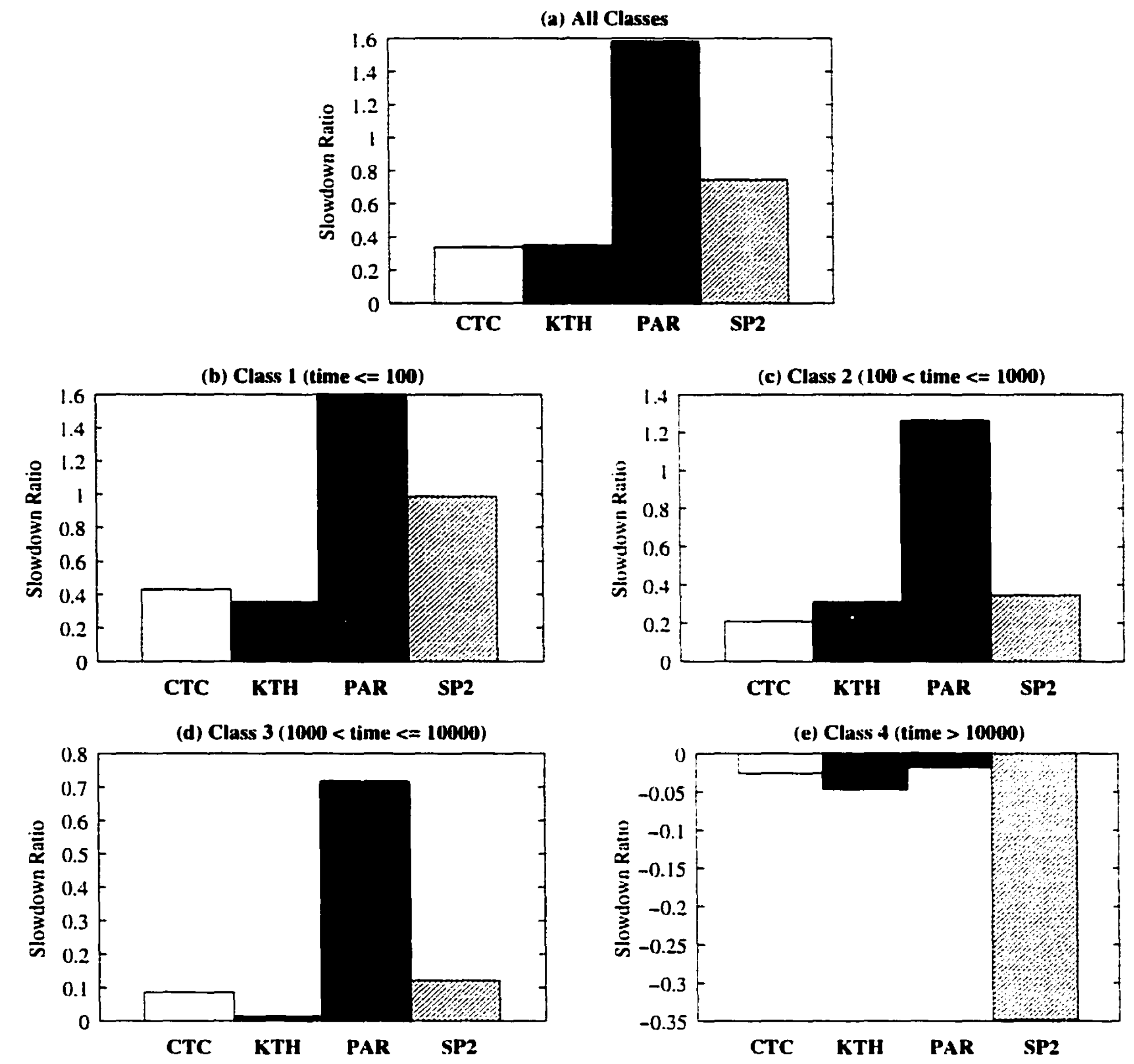


Figure 5.5: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces. Using actual service times as accurate estimates, we compare multiple-queue backfilling (with four-part classification) to single-queue backfilling, both without speculative execution.

magnitude of improvement seen in the other job classes.

Transient measures illustrate how well each policy responds to sudden arrival bursts. Furthermore, transient measures reflect the end-user perception of system performance, i.e., how well the policy performs during the relatively small window of time that the user interacts with the system. Figure 5.6 depicts transient snapshots of the slowdown ratio versus time for each of the four traces. For all traces, marked improvement (i.e., $\mathcal{R} > 0$) in slowdown is achieved using the multiple-queue backfilling policy. Although the single-queue policy gives better slowdown (i.e., $\mathcal{R} < 0$) for a relatively few batches, multiple-queue backfilling excels with more frequent and larger improvements.

5.4.2 Backfilling Policy Comparison Using Inaccurate Estimates

We now compare the performance of the two backfilling policies using user-provided inaccurate estimates. For the results in this section, we compare the performance of multiple-queue backfilling (using the three-part classification described in Section 5.2.2.2) to single-queue backfilling, both employing speculative execution.

Figure 5.7 depicts the aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling for each of the four traces. Figure 5.7(a) depicts $\mathcal{R}$ for all job classes combined, while Figures 5.7(b)–(d) each depict $\mathcal{R}$ for an individual job class. As was the case with accurate estimates, multiple-queue backfilling provides better job slowdown (i.e., $\mathcal{R} > 0$) for all classes combined in the presence of inaccurate estimates, as shown in Figure 5.7(a). With the exception of the long job class in the two SDSC workloads, multiple-queue backfilling also provides better average job slowdown within each of the individual job classes, as shown in Figures 5.7(b)–(d).

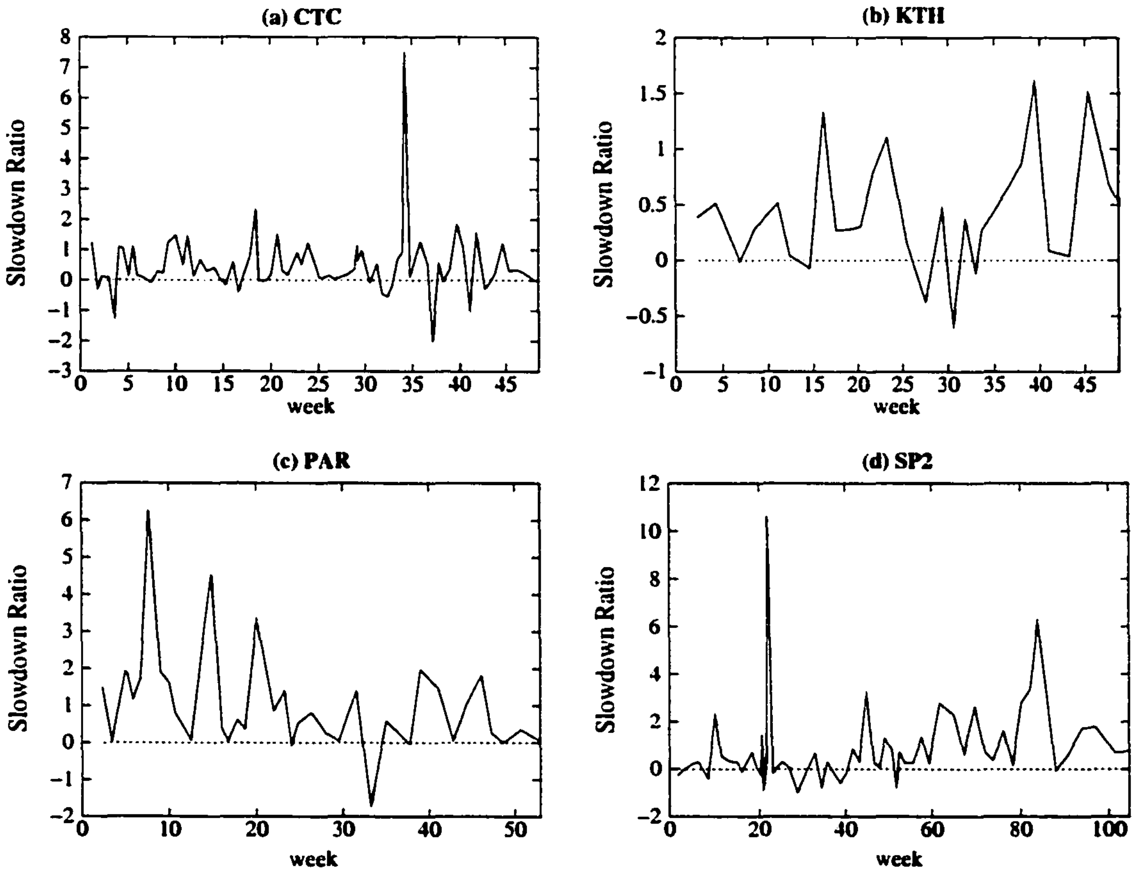


Figure 5.6: Slowdown ratio $\mathcal{R}$ per 1000-job submissions as a function of time for each of the four traces. Using actual service times as accurate estimates, we compare multiple-queue backfilling (with four-part classification) to single-queue backfilling, both without speculative execution.

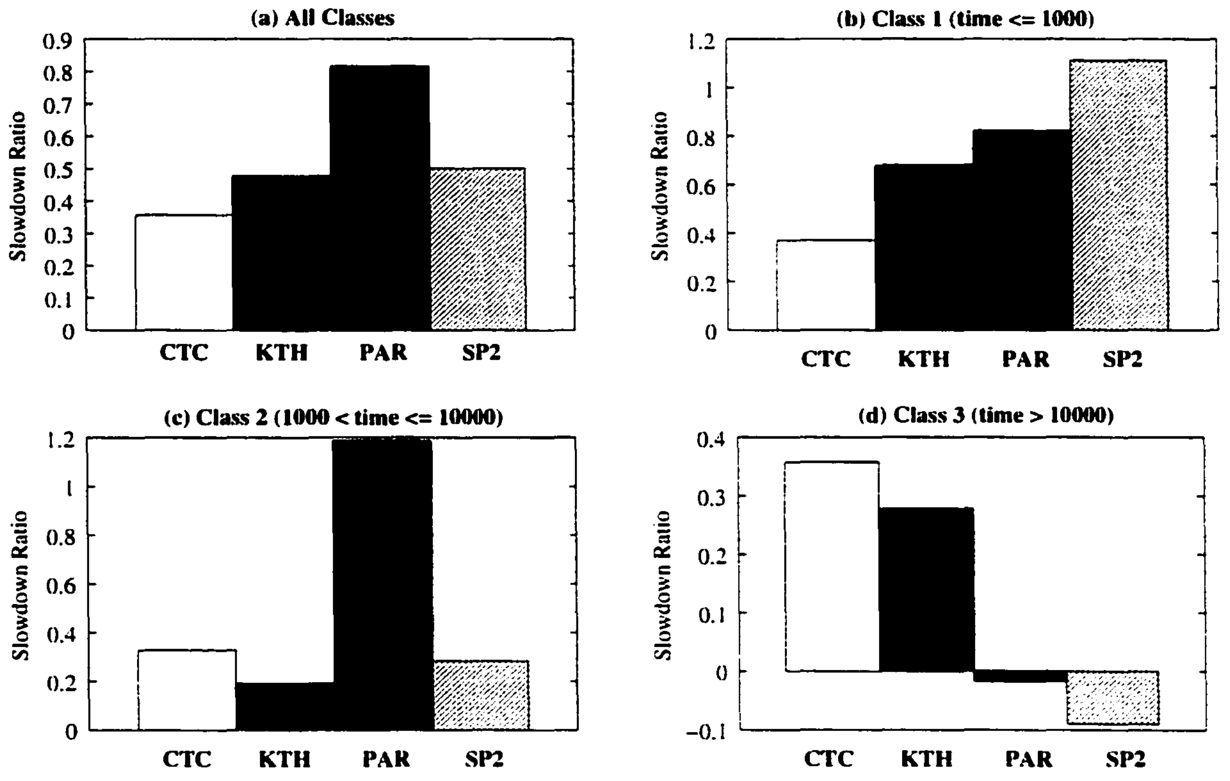


Figure 5.7: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces using inaccurate user estimates. Using user-provided inaccurate estimates, we compare multiple-queue backfilling (with three-part classification) to single-queue backfilling, both employing speculative execution.

Furthermore, Figure 5.8 depicts transient snapshots of the slowdown ratio versus time for each of the four traces. Again, noticeable improvement is achieved (i.e., $\mathcal{R} > 0$) using multiple-queue backfilling. Notice that for a few batches, single-queue backfilling gives better performance, e.g., the two large negative spikes for the CTC workload. However, across time $\mathcal{R}$ is predominantly positive for all traces, corresponding to the performance gains depicted in Figure 5.7.

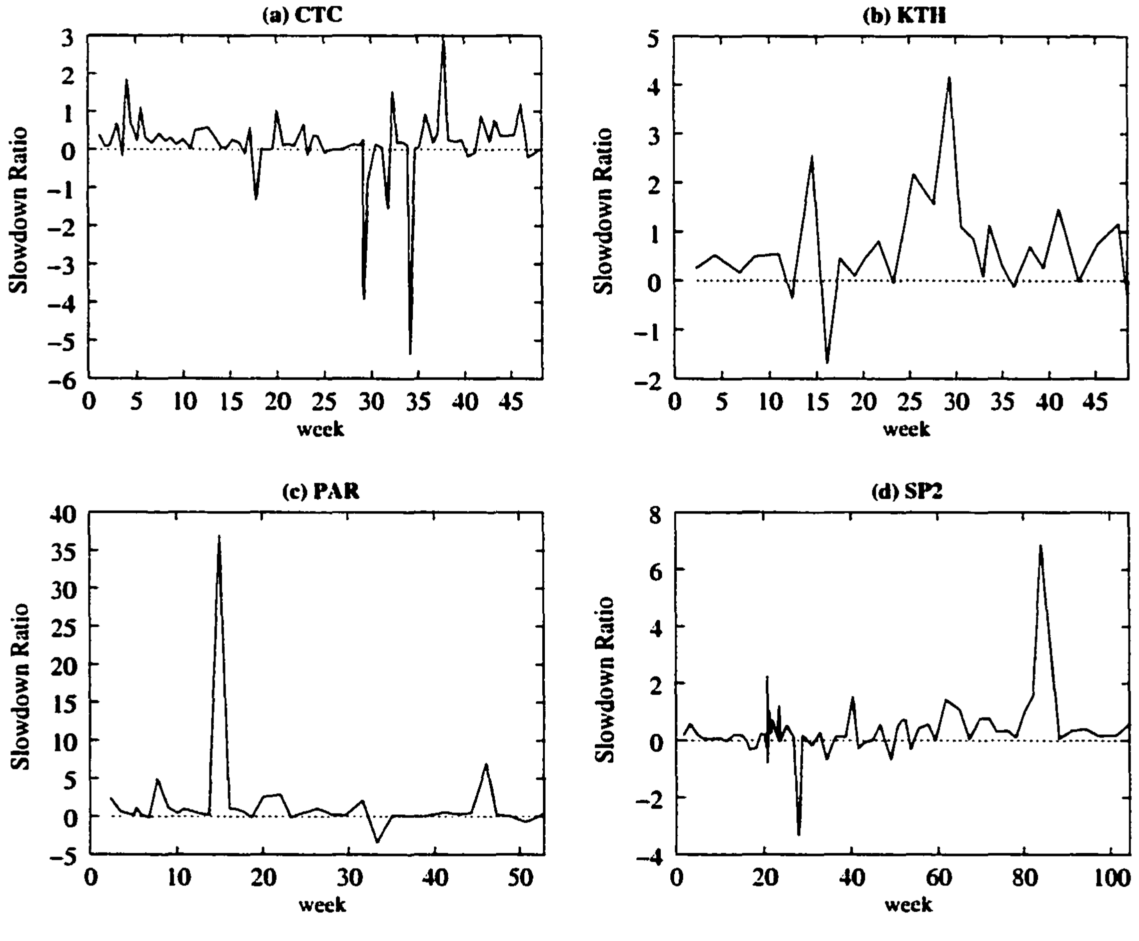


Figure 5.8: Slowdown ratio $\mathcal{R}$ per 1000-job submissions as a function of time for each of the four traces. Using user-provided inaccurate estimates, we compare multiple-queue backfilling (with three-part classification) to single-queue backfilling, both employing speculative execution.

Because multiple-queue backfilling with speculative execution performs well given poor user estimates, and because workload analysis shows that users frequently overestimate job service times, henceforth we assume inaccurate estimates. For all figures to follow, we therefore compare multiple-queue backfilling using three-part classification to single-queue backfilling, both employing speculative execution.

5.4.3 Backfilling Policy Comparison Under Heavy Load

Most scheduling policies perform well under low system load because little, if any, queueing is present. To further evaluate multiple-queue backfilling, we now consider its performance under heavy system load when scheduling is more difficult. We impose a heavier system load than that of the trace by linearly scaling (reducing) subsequent interarrival times in the trace. Effectively, we linearly increase the arrival rate of jobs into the system. Note that with this modification, we preserve the statistical characteristics of the arrival pattern in the original trace, except that the same jobs now arrive "faster".

Figure 5.9 again depicts the aggregate slowdown ratio $\mathcal{R}$ using multiple-queue backfilling relative to single-queue backfilling for each of the four workloads. For each workload, we display $\mathcal{R}$ for the original arrival rate and for arrival rates multiplied by factors of 1.25 and 1.50. In all figures, the multiple-queue and single-queue backfilling policies experience the same rate of arriving jobs.

Consistent with the results presented in Figure 5.7, multiple-queue backfilling provides better average job slowdown than single-queue backfilling for all job classes combined and for each individual job class (with the exception of the two SDSC workloads in the long job class) using the original arrival rates. When we increase the arrival rates, multiple-queue backfilling continues to provide better average job slowdown for all job classes combined

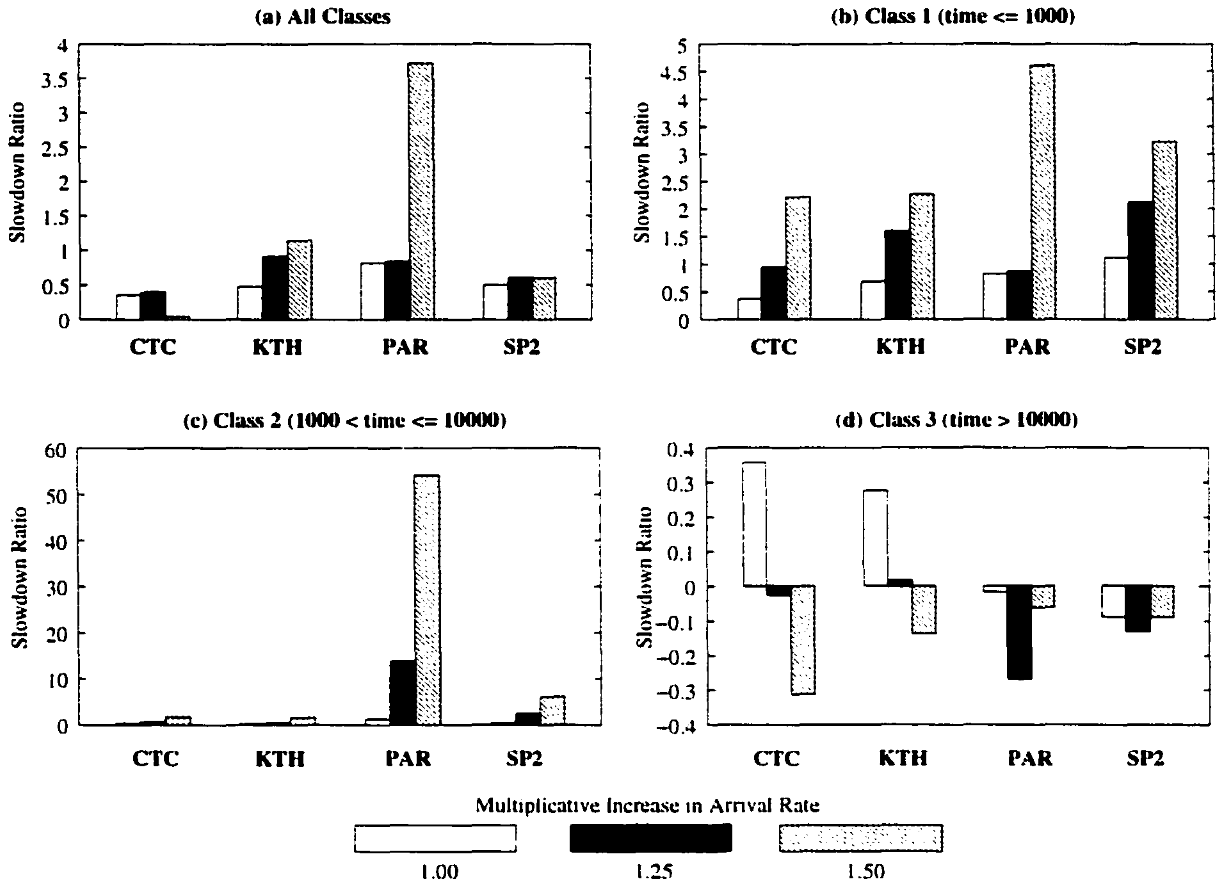


Figure 5.9: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with increasing system load. All slowdown ratios are computed relative to single-queue backfilling under the same load.

(Figure 5.9(a)) and for the small and medium job classes (Figures 5.9(b) and (c)). Notice that as system load becomes heavier, backfilling become more difficult and a corresponding decrease in performance for class three jobs results. As discussed earlier, a queued job competes directly only with other jobs in the same queue so that short jobs tend to be scheduled more quickly and long jobs tend to be delayed slightly. Therefore, multiple-queue backfilling assists shorter jobs at the expense of long jobs, and a decline in the performance of the long job class is unavoidable (Figure 5.9(d)).

5.4.4 Backfilling Policy Comparison Under Job Priorities

We now consider policy performance within the context of scheduling as part of a computational grid by incorporating job priorities. We assume that jobs have one of two possible priorities. High priority jobs are local jobs, and therefore should be executed in a timely manner. Low priority jobs are jobs originating from an external source that can be serviced when the system is not occupied with local jobs. Our goal is for multiple-queue backfilling to provide performance gains for both priorities.

We first consider a system in which a proportion $1-p = 0.25$ of the total submissions are from an external source in the computational grid, i.e., $p = 0.75$ of the total jobs are high priority. We select at random[8] $p = 0.75$ of the total jobs from the trace to be high priority jobs, so that the remaining $1 - p = 0.25$ are low priority jobs. Figure 5.10 depicts the corresponding aggregate slowdown ratio $\mathcal{R}$ of multiple-queue backfilling relative to single-queue backfilling for each of the four traces. Figure 5.10(a) shows $\mathcal{R}$ for all job classes

[8]The jobs selected at random are determined by the random number sequence as manifested by the choice of initial seed. While different initial seeds give slightly different results, the qualitative analysis remains the same — multiple-queue backfilling consistently outperforms single-queue backfilling.

combined, while Figures 5.10(b)–(d) each show $\mathcal{R}$ for an individual job class. For each trace, we also provide $\mathcal{R}$ as computed for high priority jobs, for low priority jobs, and for both priorities combined. As shown in Figure 5.10(a), multiple-queue backfilling provides better average job slowdown than single-queue backfilling for all job classes combined. Also note that, with the exception of the long job class depicted in Figure 5.10(d), multiple-queue backfilling tends to perform better within each of the individual job classes. Again, because multiple-queue backfilling assists shorter jobs at the expense of long jobs, a decline in the performance of the long job class is unavoidable.

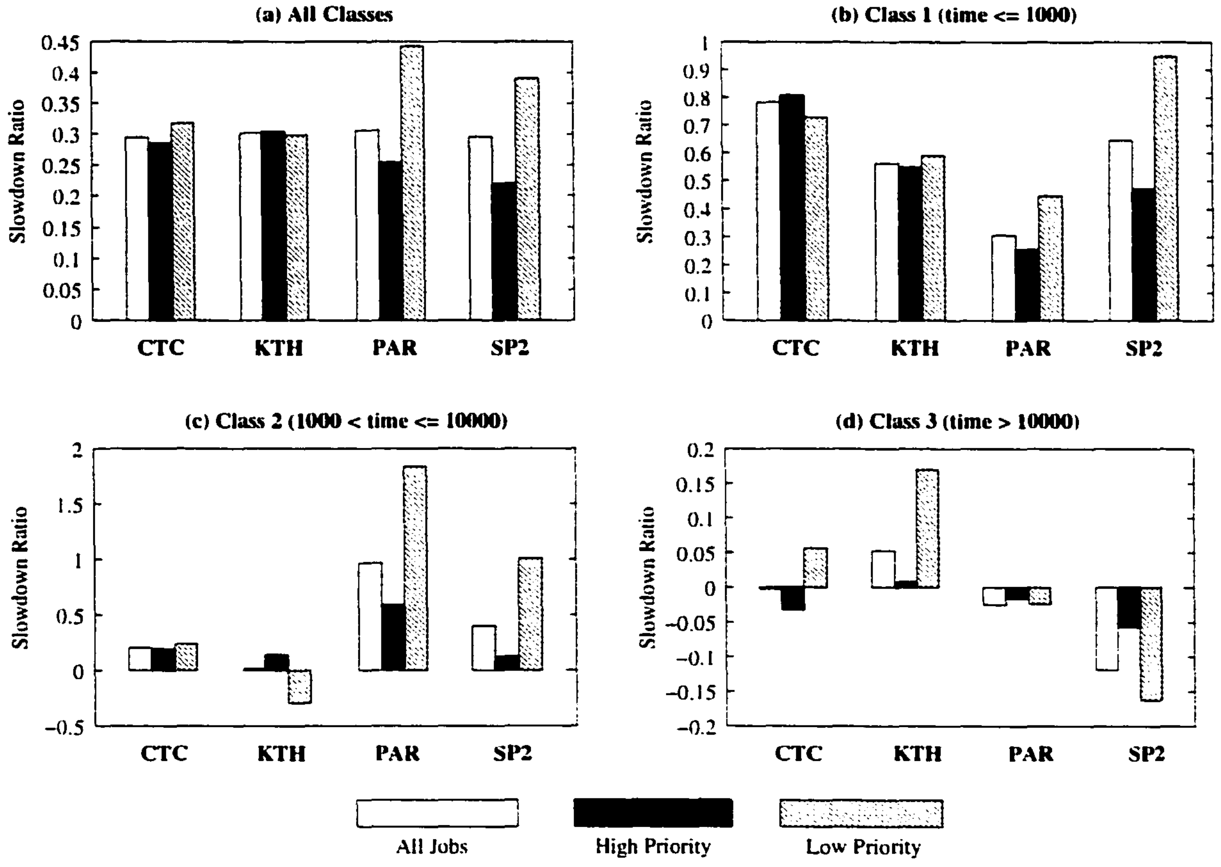


Figure 5.10: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces where $p = 0.75$ of the total jobs have high priority

Notice that in many cases, the performance gains for low priority jobs are of greater magnitude than the gains for high priority jobs. These differences in magnitude are attributable to the fact that, because high priority jobs are given preferential treatment, both the multiple- *and* single-queue policies assist high priority jobs. The single-queue policy causes low priority jobs to suffer as a result of assisting the high priority jobs. However, because the multiple-queue policy effectively partitions jobs, multiple-queue backfilling improves slowdown for *both* priorities.

Because a system that is part of a computational grid can experience dramatic changes in workload across time, we also consider the transient performance of multiple-queue versus single-queue backfilling under job priorities. Figure 5.11 depicts transient snapshots of the slowdown ratio versus time for each of the four traces where $p = 0.75$ of the total jobs have high priority. Each figure shows slowdown ratio snapshots for high priority jobs and low priority jobs. Again, marked improvement in slowdown is achieved ($\mathcal{R} > 0$) using multiple-queue backfilling. Although single-queue backfilling provides better slowdown ($\mathcal{R} < 0$) for a few batches, $\mathcal{R}$ is positive for a majority of the batches corresponding to performance gains using multiple-queue backfilling.

We also consider a system in which only $1 - p = 0.05$ of the submissions are external, i.e., $p = 0.95$ of the total jobs have high priority. Figures 5.12 and 5.13 are analogous to Figures 5.10 and 5.11, except $p = 0.95$ of the total jobs have high priority. Again, we see that multiple-queue backfilling improves average job slowdown for all job classes combined and, with the exception of long jobs in the two SDSC workloads, for the individual job classes. Also note the larger vertical axis scales in Figure 5.12 corresponding to even larger performance gains than when $p = 0.75$ of the total jobs have high priority.

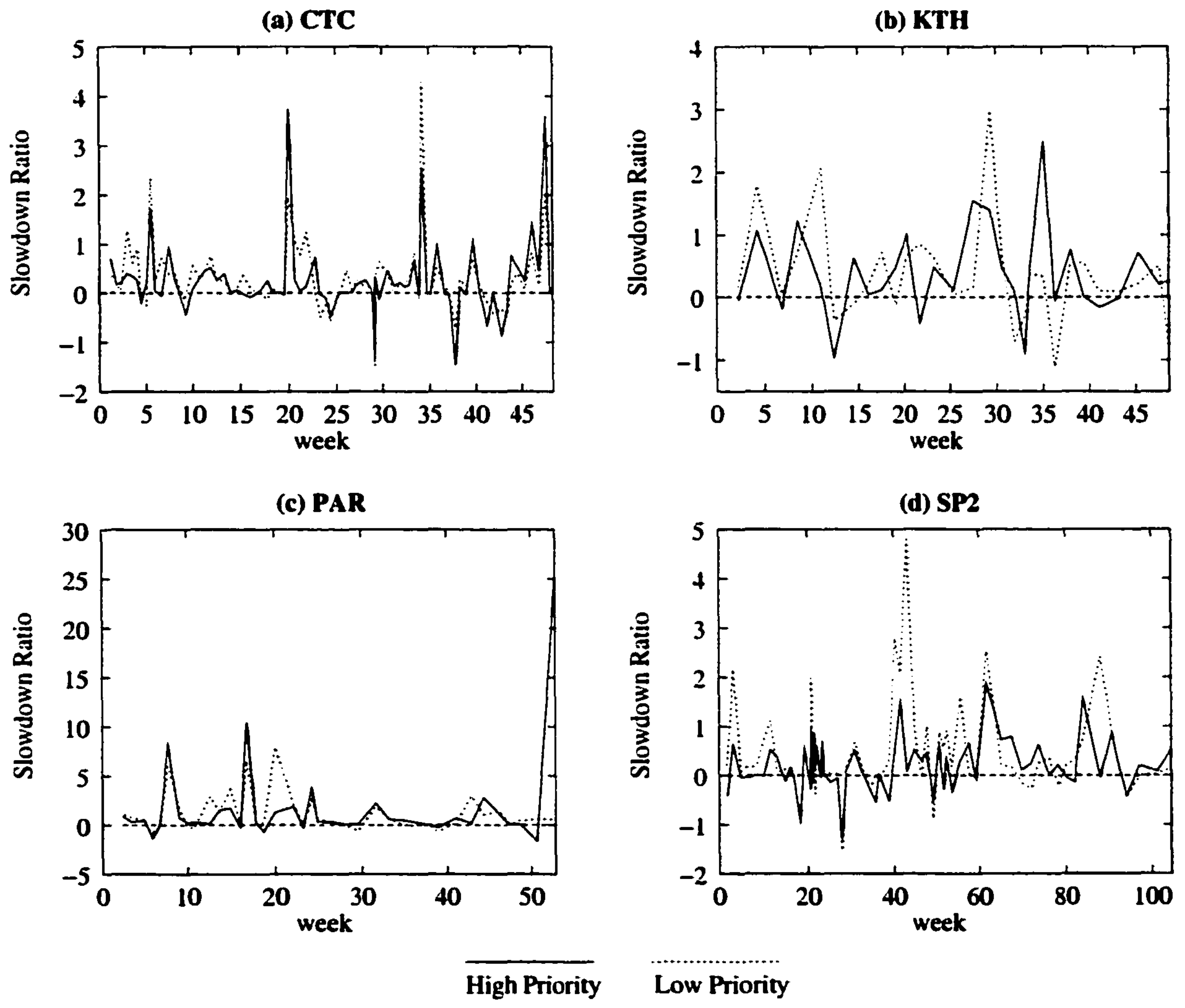


Figure 5.11: Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces where $p = 0.75$ of the total jobs have high priority

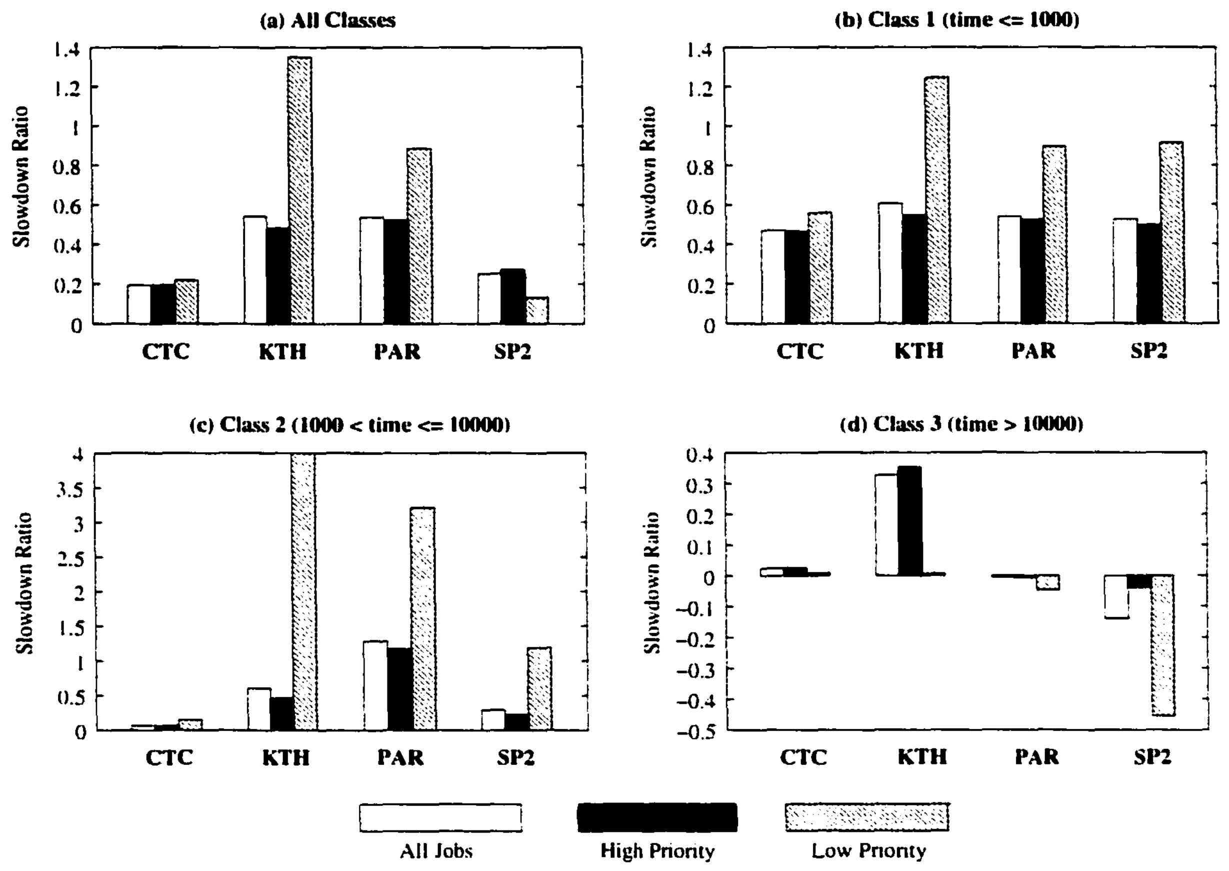


Figure 5.12: Overall and per-class aggregate slowdown ratio $\mathcal{R}$ for each of the four traces where $p = 0.95$ of the total jobs have high priority

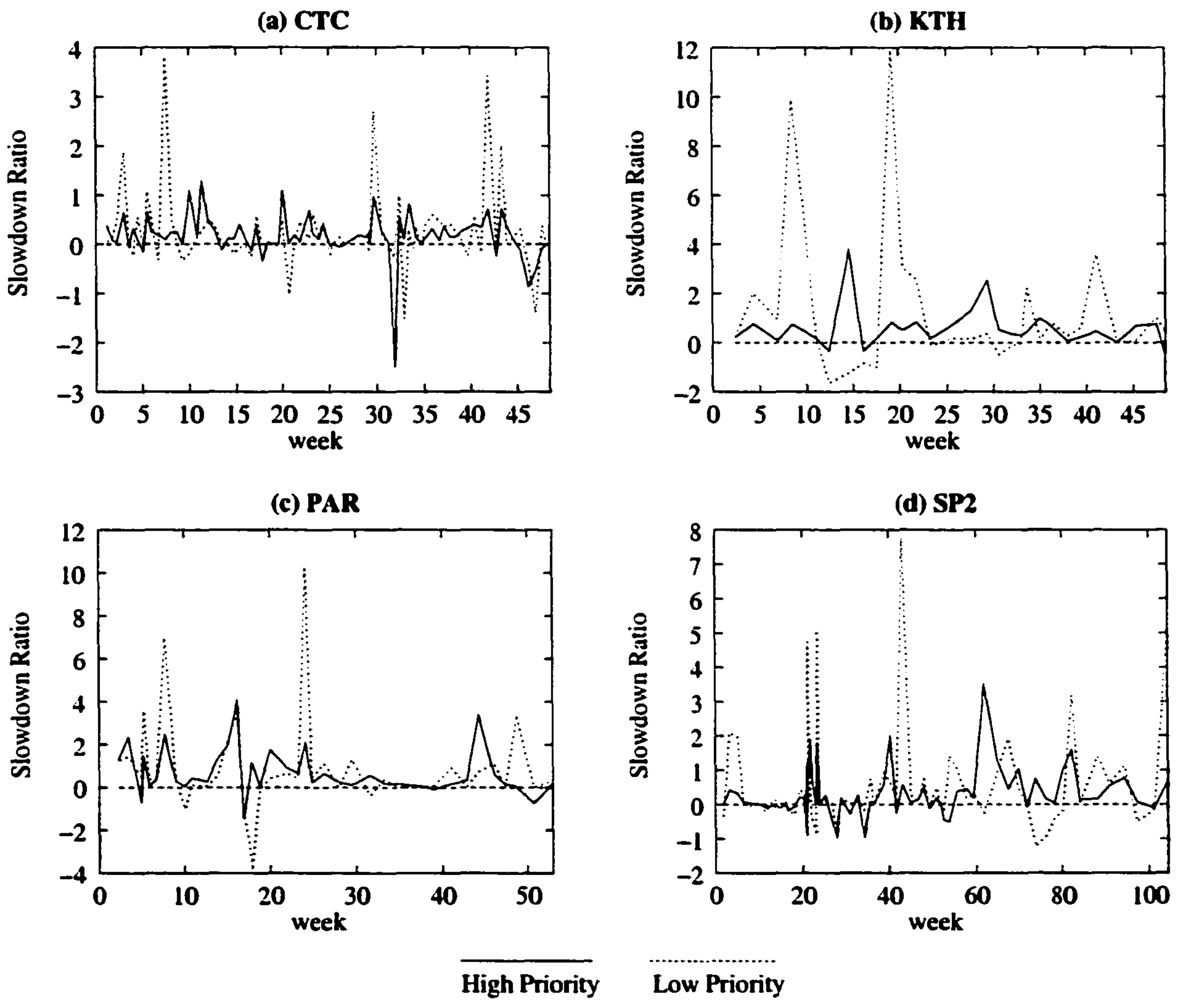


Figure 5.13: Slowdown ratio $\mathcal{R}$ per 1000 job submissions as a function of time for high priority and low priority jobs for each of the four traces where $p = 0.95$ of the total jobs have high priority

5.4.5 Backfilling Policy Comparison Under Reservations

We further evaluate policy performance for scheduling within a computational grid by incorporating reservation requests. For each of the four traces, Figure 5.14 depicts the average job slowdown for all classes combined with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations. As shown, multiple-queue backfilling provides better average job slowdown for proportions of 0.01 and 0.05, and provides comparable slowdown for a proportion of 0.25.

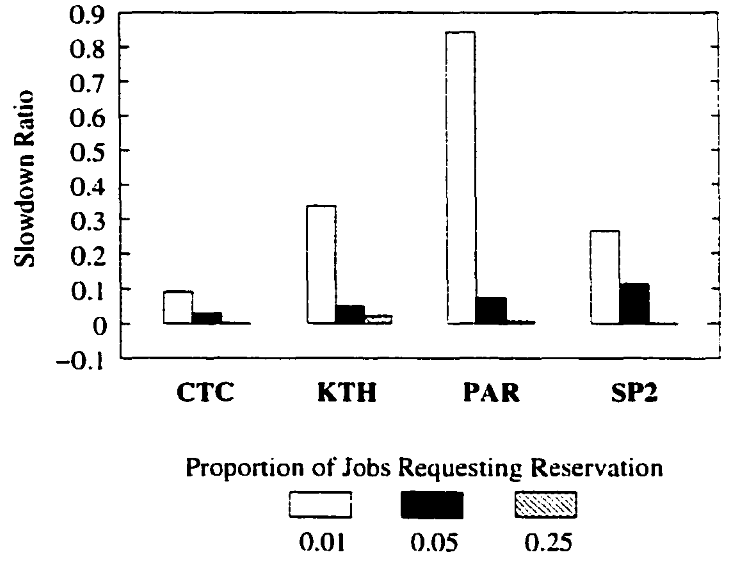


Figure 5.14: Overall aggregate slowdown ratio $\mathcal{R}$ for each of the four traces with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations

In addition, Table 5.3 shows the number of missed reservations for single-queue and multiple-queue backfilling for each of the four traces with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations. Note that both policies miss roughly the same number of reservations. For a proportion of 0.25 of the total jobs requesting reservations, Figure 5.15 depicts for each trace the tail of the distribution of delays experienced by jobs requesting reservations. As shown, multiple-queue and single-queue backfilling achieve

Workload	Proportion of Reservations	Number of Reservations	Single-Queue Missed	Multiple-Queue Missed
CTC	0.01	761	19	13
	0.05	3908	90	90
	0.25	19 897	1396	1441
KTH	0.01	273	9	12
	0.05	1421	45	38
	0.25	7178	567	546
PAR	0.01	374	5	1
	0.05	1873	7	9
	0.25	9543	138	119
SP2	0.01	652	54	46
	0.05	3349	294	250
	0.25	16 968	4051	3983

Table 5.3: Number of missed reservations for single- and multiple-queue backfilling with proportions of 0.01, 0.05, and 0.25 of the total jobs requesting reservations

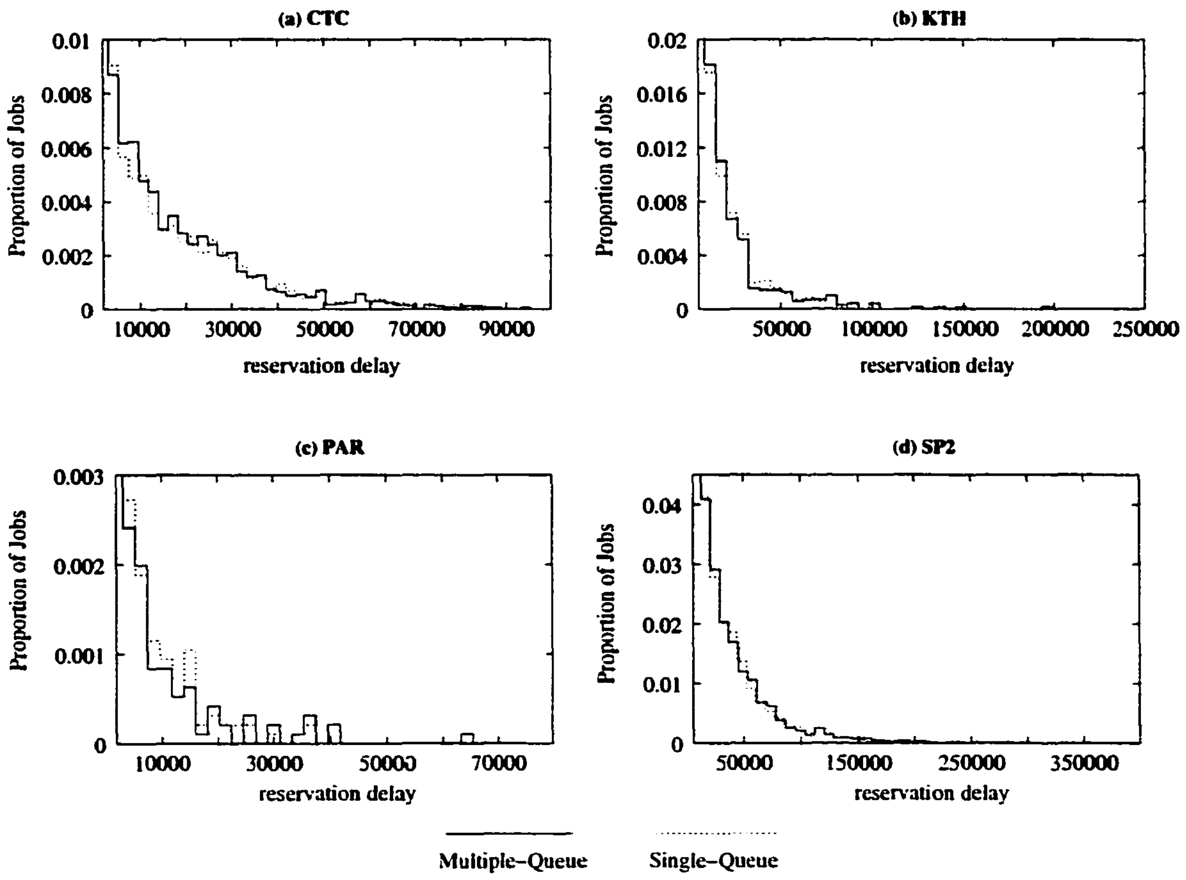


Figure 5.15: Distribution tails of the delays experienced by jobs requesting reservations with a proportion of 0.25 of the total jobs requesting reservations

roughly the same distribution for reservation delays. Although we cannot claim significant improvement relative to the number of missed reservations and the distribution of reservation delays, multiple-queue backfilling performs as well as single-queue backfilling.

5.4.6 Policy Comparison Summary

The comparison results in the previous sections provide strong evidence that the multiple-queue backfilling policy outperforms the standard single-queue backfilling policy. In the case of accurate user estimates, our four-part classification alone is sufficient for the multiple-queue policy to achieve better average job slowdown. In the case of inaccurate estimates, our three-part classification coupled with speculative execution of jobs provides better slowdown. In addition, we have shown that multiple-queue backfilling performs even better under heavy system load. We have also shown multiple-queue backfilling to be a robust scheduling policy for systems within a computational grid, providing improved slowdown under job priorities and reservations. Consequently, because of its ability to automatically adapt to changing workload conditions, because of its robustness, and because of its evident performance benefits, we claim multiple-queue backfilling to be superior to standard single-queue backfilling.

Chapter 6

Summary and Future Work

In this dissertation, we presented work based on an artificial society simulation model. In this model, new rules can be easily added and modified to simulate a variety of behaviors. By systematically perturbing the set of input parameters, we can scientifically evaluate the output that results from carefully controlled initial conditions. Hence, this area of study contains an essentially unlimited number of research topics of interest. However, in the work presented here, we restricted ourselves to the study of time evolution of the model and the corresponding simulation issues that arise as a result of the implementation.

In Chapter 2, we carefully defined a representative artificial society simulation model based on a model taken from the social sciences literature. For the sake of simplicity, we included only a subset of the possible rules, but modified the included rules to be more realistic. We also described in detail the two options available for time evolution of the model: synchronous and asynchronous time evolution.

In Chapter 3, we provided results from our artificial society simulation model showing that very different behavior can result based solely on the choice of asynchronous or synchronous time evolution. Through systematic experimentation, we showed that different behavior can be observed for a variety of model parameters and initial conditions.

In Chapter 4, we showed that the next-event simulation approach required to incorporate asynchronous time evolution into the model can lead to prohibitive execution times unless implemented properly. Based on other authors' suggestions from previous event list and priority queue research, we presented results showing that acceptable computational performance can be achieved, and made suggestions for appropriate event list implementations for artificial society models.

In Chapter 5, we showed that, like the artificial society model, scheduling resources in a parallel system also benefits by transforming from a synchronous to an asynchronous system. We described in detail a new multiple-queue backfilling scheduling policy that partitions the system based on job service estimates. We presented results showing that our policy outperforms the standard single-queue policy using accurate or inaccurate estimates. Our policy also excels in the presence of job priorities, and is robust even with job reservations.

In conclusion, the following topics are potential extensions to the work presented in this dissertation.

- Parallel simulation: Because of the limited computing and memory constraints of uniprocessor machines, parallel simulation of artificial society models is required for very large models. A direct consequence of parallelization is improved execution time. Previous work in this area concentrated on distributed memory implementations. We would like to explore parallelization in the context of shared memory and study the potential speedups for our artificial society simulation model.

- Other models of interest: We would also like to explore the topics discussed in this dissertation within artificial society models from other applications. Because of the popularity of agent-based simulation, there are many interesting models from which we can choose in order to examine the effects of time evolution and associated simulation issues.

- Extend multiple-queue backfilling: Additional avenues of study in this area include priority aging of jobs and further evaluation of the job partitioning policy and its parameters. We would also like to investigate multiple-queue backfilling in systems of heterogeneous processors and clusters of processors.

Appendix A

Random Variable Models

The material in this appendix, taken directly from [82], summarizes those continuous and discrete *random variable* models used in our artificial society discrete event simulation model. For each random variable X, we define the set of possible values of X, the probability density function (pdf) of X, and the mean and standard deviation of X[1]. The code used to generate realizations of these random variables is provided at the end of this appendix.

A.1 Uniform

The continuous random variable X is *Uniform* (a, b) if

- the real-valued parameters a, b satisfy $a < b$
- the possible values of X are $\mathcal{X} = \{x \mid a < x < b\}$
- the probability density function of X is

$$f(x) = \frac{1}{b-a} \qquad a < x < b$$

[1]In the definitions to follow, the use of μ to represent the mean and the use of σ to represent the standard deviation is consistent with conventional statistical notation and is not to be confused with the μ and σ notations from Chapter 2.

as illustrated in Figure A.1

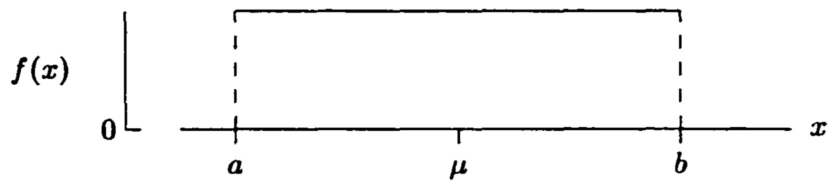


Figure A.1: Probability density function of a *Uniform*(a, b) random variable

- the mean of X is

$$\mu = \frac{1}{2}(a+b)$$

- the standard deviation of X is

$$\sigma = \frac{b-a}{\sqrt{12}}$$

A.2 Exponential

The continuous random variable X is *Exponential*(μ) if

- the real-valued parameter μ satisfies $\mu > 0$
- the possible values of X are $\mathcal{X} = \{x \mid x > 0\}$
- the probability density function of X is

$$f(x) = \frac{1}{\mu}\exp(-x/\mu) \qquad x > 0$$

as illustrated in Figure A.2

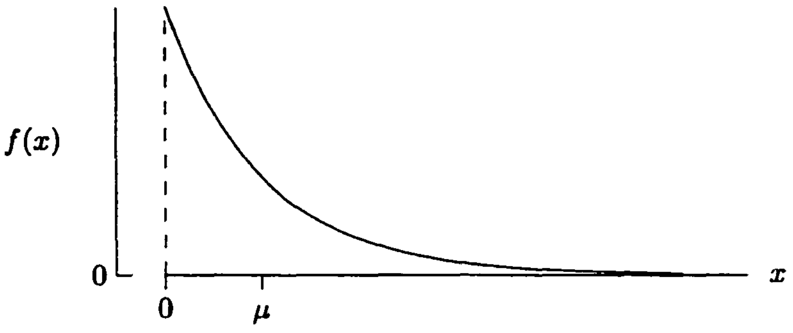


Figure A.2: Probability density function of an *Exponential*(μ) random variable

- the mean of X is μ
- the standard deviation of X is

$$\sigma = \mu$$

A.3 Equilikely

The discrete random variable X is *Equilikely*(a, b) if if

- the parameters a, b are integers with $a < b$
- the possible values of X are $\mathcal{X} = \{a, a+1, \ldots, b\}$
- the probability density function of X is

$$f(x) = \frac{1}{b-a+1} \qquad x = a, a+1, \ldots, b$$

  as illustrated in Figure A.3

- the mean of X is

$$\mu = \frac{a+b}{2}$$

- the standard deviation of X is

$$\sigma = \sqrt{p(1-p)}$$

A.5 Random Variate Generator Code

A random *variate* is an algorithmically generated realization of a random *variable*. Here we provide C code, written by Steve Park and David Geyer [82], to generate random *variates* of the corresponding random *variable* models presented above. Each random variate generator makes use of a function `Random()` that produces a real-valued number between 0.0 and 1.0. `Random()` is an implementation of a Lehmer generator for architectures that support 32-bit two's complement arithmetic. In addition, by using the function `SelectStream()`, each stochastic component in the artificial society simulation model is allocated one of 256 disjoint streams of random numbers. Such disjoint streams provide a unique source of randomness for each stochastic component. The function `PlantSeeds()` can be used to set the initial random number seed for the collection of streams.

```
#define MODULUS    2147483647
#define MULTIPLIER 48271
#define CHECK      399268537
#define STREAMS    256        /* # of streams                        */
#define A256       22925      /* jump multiplier                     */
#define DEFAULT    123456789  /* initial seed, use 0 < DEFAULT < MODULUS */

static long seed[STREAMS] = {DEFAULT}; /* current state of each stream   */
static int  stream        = 0;         /* stream index, 0 is the default */
static int  initialized   = 0;         /* test for stream initialization */
```

```
  double Random(void)
/* =================================================================
 * Random returns a pseudo-random real number uniformly distributed
 * between 0.0 and 1.0.
 * ================================================================= */
{
  const long Q = MODULUS / MULTIPLIER;
  const long R = MODULUS % MULTIPLIER;
        long t;

  t = MULTIPLIER * (seed[stream] % Q) - R * (seed[stream] / Q);
  if (t > 0)
    seed[stream] = t;
  else
    seed[stream] = t + MODULUS;
  return ((double) seed[stream] / MODULUS);
}

  void PlantSeeds(long x)
/* =================================================================
 * Use this (optional) function to set the state of all the random
 * number generator streams by "planting" a sequence of states (seeds),
 * one per stream, with all states dictated by the state of the default
 * stream.  The sequence of planted states is separated one from the
 * next by 8,367,782 calls to Random(),
 * ================================================================= */
{
  const long Q = MODULUS / A256;
  const long R = MODULUS % A256;
        int  j;
        int  s;

  initialized = 1;
  s = stream;                          /* remember the current stream */
  SelectStream(0);                     /* change to stream 0          */
  PutSeed(x);                          /* set seed[0]                 */
  stream = s;                          /* reset the current stream    */
  for (j = 1; j < STREAMS; j++) {
    x = A256 * (seed[j - 1] % Q) - R * (seed[j - 1] / Q);
    if (x > 0)
      seed[j] = x;
    else
      seed[j] = x + MODULUS;
   }
}
```

```
  void SelectStream(int index)
/* =====================================================================
 * Use this (optional) function to set the current random number generator
 * stream -- that stream from which the next random number will come.
 * =====================================================================
 */
{
  if ((initialized == 0) && (index != 0)) {  /* protect against        */
    initialized = 1;                         /* un-initialized streams */
    PlantSeeds(DEFAULT);
  }
  stream = ((unsigned int) index) % STREAMS;
}

   double Uniform(double a, double b)
/* ==========================================================
 * Returns a uniformly distributed real number between a and b.
 * NOTE: use a < b
 * ==========================================================
 */
{
  return (a + (b - a) * Random());
}

   double Exponential(double m)
/* =========================================================
 * Returns an exponentially distributed positive real number.
 * NOTE: use m > 0.0
 * =========================================================
 */
{
  return (-m * log(1.0 - Random()));
}

   long Equilikely(long a, long b)
/* ==================================================================
 * Returns an equilikely distributed integer between a and b inclusive.
 * NOTE: use a < b
 * ==================================================================
 */
{
  return (a + (long) ((b - a + 1) * Random()));
}
```

```
   long Bernoulli(double p)
/* ========================================================
 * Returns 1 with probability p or 0 with probability 1 - p.
 * NOTE: use 0.0 < p < 1.0
 * ========================================================
 */
{
  return ((Random() < (1.0 - p)) ? 0 : 1);
}
```

Appendix B

Related Simulation Models

In this appendix, we provide results from models similar to our artificial society model to further support our claim that the time evolution of a model can have a dramatic effect on the output. We replicate results from two different cellular automata based models *without* agents to demonstrate that, similar to time evolution in the artificial society model, very different results in output can be obtained by updating cells in an asynchronous, rather than a synchronous, manner.

B.1 Soil Erosion Model

We consider the soil erosion model described in Section 9.5 of [105] used to study the relationship between land development and erosion occurring as a result of the development. The goal is to deterministically develop as much land as possible without causing the land to collapse by erosion from over-development.

The landscape is a two-dimensional $X \times Y$ cellular automaton in which each (x, y) cell represents a plot of land. Each cell in the landscape can be in exactly one of two possible states, either *undeveloped* or *developed*. An undeveloped cell may become developed, but once a cell has become developed, it cannot revert back to an undeveloped state, i.e., there is no land reclamation in the model.

Two separate processes may cause an undeveloped cell to become developed. One of these processes simulates the intentional development of a landscape cell by human action. The second process simulates the natural erosion of a landscape cell that results from over-development of neighboring landscape cells by humans. To model each process, a deterministic rule is applied on a cell-by-cell basis across the landscape. The *develop rule*, which simulates intentional land development, and the *stabilize rule*, which simulates erosion, are defined as follows.

- *Stabilize Rule*: An undeveloped cell is considered "stable" and will remain undeveloped provided there is an undeveloped cell within the nine-cell Moore neighborhood somewhere to the north (any of the northwest, north or northeast positions), somewhere to the south, somewhere to the west, and somewhere to the east; otherwise the undeveloped cell is considered "loose" and will become developed as a result of erosion.

- *Develop Rule*: An undeveloped cell is considered safe for intentional development provided there is an undeveloped cell at each position to the immediate north, south, west, and east; otherwise the undeveloped cell cannot be developed.

We adopt periodic boundary conditions so that each cell is the center of its own nine-cell Moore neighborhood.

Figure B.1 shows the effect of the *stabilize* rule on the center cell for three nine-cell Moore neighborhood examples. Undeveloped cells are white; developed cells are shaded.

The undeveloped center cells in Figure B.1(a) and (b) are stable and will therefore remain undeveloped when the stabilize rule is applied to the landscape. Because there is no undeveloped cell to the west, the undeveloped center cell in Figure B.1(c) is unstable and will become developed (i.e., erode) when the stabilize rule is applied.

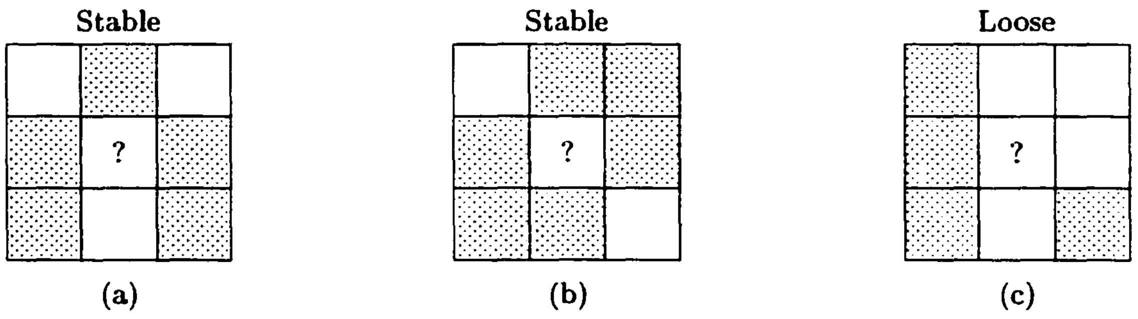


Figure B.1: Effect of stabilize rule on center cell

Figure B.2 shows the effect of the ***develop*** rule on the center cell for two nine-cell Moore neighborhood examples. The undeveloped center cell in Figure B.2(a) is considered safe for development and will become a developed cell when the develop rule is applied to the landscape. Because there is a developed cell to the immediate north, the undeveloped center cell in Figure B.2(b) is unsafe for development and will remain an undeveloped cell when the develop rule is applied.

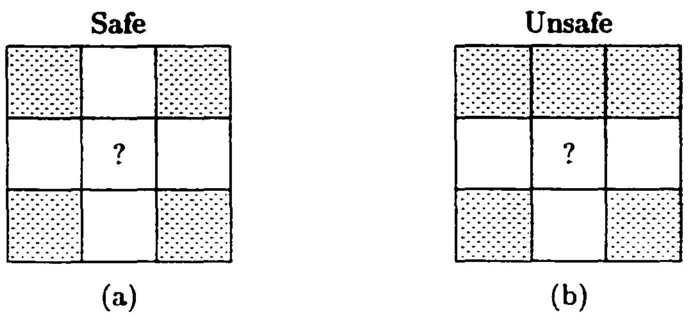


Figure B.2: Effect of develop rule on center cell

All cells are initially undeveloped except for a small proportion p of cells selected at random that are developed. The evolution of the landscape occurs as a three-step process.

- The landscape is stabilized via repeated application of the stabilize rule until there is no change in the state of any cell.
- Provided there are remaining undeveloped cells, the develop rule is applied to the landscape to deterministically develop as many undeveloped cells as possible.
- The landscape is again stabilized via repeated application of the stabilize rule until there is no change in the state of any cell.

The final stabilization step causes the landscape to either achieve a stable configuration with $p < 1.0$, or to completely erode, in which case $p = 1.0$.

The develop rule can be applied to the landscape either synchronously or asynchronously[1]. In this context, synchronous updating involves applying the rule to all cells simultaneously in parallel so that a change in the state of any cell is not realized by other cells. Two copies of the landscape are maintained to achieve this parallelism. One copy is the landscape before the develop rule is applied. The decision of whether to develop a cell is based on state information of the cells using this copy. The second copy contains the updated state information of the cells after the develop rule is applied. For an $X \times Y$ landscape, synchronous application of the develop rule is shown in Algorithm B.1.

Asynchronous updating is characterized by applying the develop rule in sequence to (x, y) cells selected at random, without replacement. For each application of the develop rule to an (x, y) cell, the landscape is appropriately updated so that subsequently selected

[1]Because the stabilize rule is applied repeatedly until no cell changes state, the discussion of asynchronous or synchronous application is not relevant to the stabilize rule.

```
while ( no change in the state of any cell )
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                Stabilize($x$, $y$);

for ($x = 0$; $x < X$; $x$++)
        for ($y = 0$; $y < Y$; $y$++)
            Develop($x$, $y$);   /* in parallel using two landscape copies */

while ( no change in the state of any cell )
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                Stabilize($x$, $y$);
```

Algorithm B.1: Develop Rule algorithm — synchronous application

cells will recognize any change in state from previously selected cells. Therefore, only one copy of the landscape is required for asynchronous application of the develop rule. For an $X \times Y$ landscape, asynchronous application of the develop rule is Algorithm B.2.

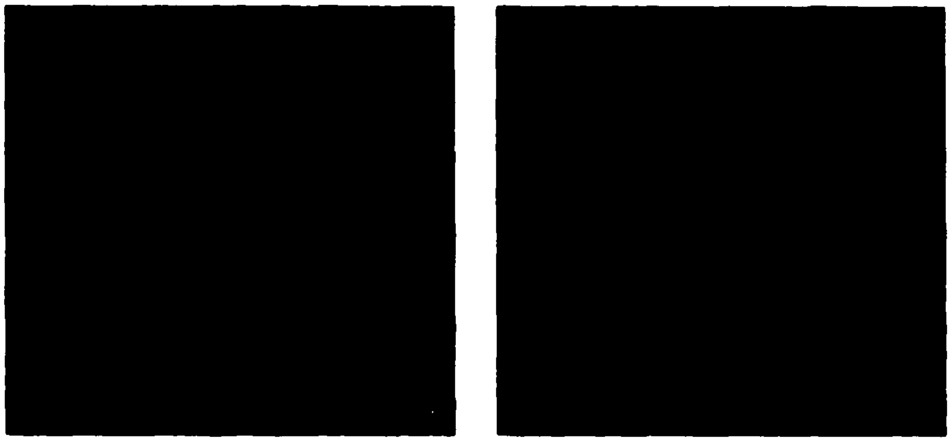
```
while ( no change in the state of any cell )
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                Stabilize($x$, $y$);

while ( there remain cells to be selected ) {
        select an $(x, y)$ cell at random, without replacement;
        Develop($x$, $y$);   /* in sequence using one landscape copy */
}

while ( no change in the state of any cell )
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                Stabilize($x$, $y$);
```

Algorithm B.2: Develop Rule algorithm — asynchronous application

Replicating the results from [105] verifies that very different output can be observed if the develop rule is applied asynchronously rather than synchronously. As a specific example, consider an $X \times Y = 50 \times 50$ landscape with $p = 0.17$ initially developed cells selected at random, as shown in Figure B.3(a)[2]. This initial configuration results in $p = 0.19$ developed cells after the first application of the stabilize rule, as shown in Figure B.3(b).



(a) Initial $p = 0.17$ development (b) $p = 0.19$ after first stabilize rule

Figure B.3: Example 50 x 50 landscape to be developed

Synchronous application of the develop rule to the landscape in Figure B.3(b) produces $p = 0.58$ developed cells, and when the stable rule is reapplied, the landscape completely erodes as illustrated in Figure B.4. However, asynchronous application of the develop rule produces $p = 0.39$ developed cells, and when the stable rule is reapplied, the landscape maintains a stable configuration of $p = 0.39$ developed cells as illustrated in Figure B.5. Admittedly, this is an extreme example, but it emphasizes our claim that results can be very sensitive to the choice of synchronous or asynchronous updating[3].

[2]In Figures B.3 – B.5, undeveloped cells are green and developed cells are black.

[3]For both synchronous and asynchronous application of the develop rule, the final proportion of developed cells is dependent on the random variate sequence that yields the initial proportion of developed cells, as determined by the choice of the initial seed. For asynchronous application, the final proportion of developed cells is also dependent on the random variate sequence of (x, y) cells selected for updating, also determined by the choice of initial seed. We use 123456789 as the initial seed for the results shown. We observe nearly identical results for other initial seeds.

(a) $p = 0.58$ after develop rule

(b) $p = 1.00$ after final stabilize rule

Figure B.4: Synchronous application of the develop rule to Figure B.3(b)

(a) $p = 0.39$ after develop rule

(b) $p = 0.39$ after final stabilize rule

Figure B.5: Asynchronous application of the develop rule to Figure B.3(b)

B.2 Prisoners' Dilemma Model

We now consider the long-standing Prisoners' Dilemma model [3] for studying the evolution of cooperative behavior. As in [80], we extend the model to two dimensions by representing the landscape as an $X \times Y$ cellular automaton. In this model, each (x, y) cell assumes one of two possible states, either *cooperator* or *defector*. The dilemma is an encounter between two neighboring cells in which payoffs are awarded according to the states of the cells. The payoff p to cell (x_1, y_1) interacting with cell (x_2, y_2) is defined by Table B.1, where T, R, P, and S are parameters with $S \leq P < R < T$.

state of (x_1, y_1)	state of (x_2, y_2)	p
defector	cooperator	T
cooperator	cooperator	R
defector	defector	P
cooperator	defector	S

Table B.1: Payoff p to cell (x_1, y_1) encountering cell (x_2, y_2)

Similar to the soil erosion model, the encounter between cells can be applied in a synchronous or asynchronous manner. The synchronous application of the Prisoners' Dilemma proceeds as follows. Each cell totals the payoffs from interactions with each of the cells in its nine-cell Moore neighborhood, including an interaction with itself. Once all cells have accumulated their payoffs, the state of each (x, y) cell then becomes the state of the cell with the largest *total* payoff within (x, y)'s neighborhood. This sequence of events, constituting one time step, continues indefinitely or until there is no change in the landscape. For an $X \times Y$ landscape, the dilemma evolves synchronously according to Algorithm B.3.

To replicate the results from [80], we use non-periodic boundary conditions on an $X \times$

```
while ( no change in the state of any cell ) {
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                SumPayoffs($x$, $y$);

        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                EvaluateState($x$, $y$);    /* in parallel using two landscape copies */
}
```

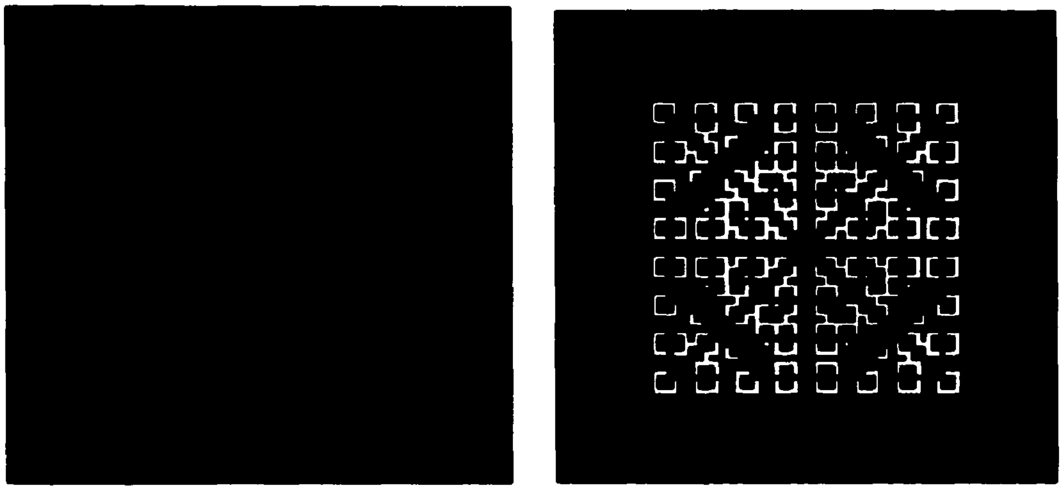
Algorithm B.3: Prisoners' Dilemma algorithm — synchronous application

$Y = 99 \times 99$ landscape so that cells on the border have fewer neighbors than those in the interior. A nine-cell Moore neighborhood is used, and the payoff parameters are $S = P = 0$, $R = 1$, and $T = 1.9$.

If the landscape is updated synchronously, two copies of the landscape are required to apply the dilemma to all cells simultaneously in parallel. Any change in the state of a cell does not affect other cells until the next time step. Starting with an initial configuration of a sole defector in the middle surrounded by all cooperators, as shown in Figure B.6(a), an "evolutionary kaleidoscope" evolves after $t = 30$ time steps, as shown in Figure B.6(b)[4]. Successive time steps continue to produce varied symmetrical patterns.

Asynchronous application of the Prisoners' Dilemma is characterized by applying the dilemma in sequence to (x, y) cells selected at random, without replacement. At time $t = 0$, initial payoffs are computed for each cell in the landscape. An (x, y) cell is then selected at random, and its state becomes the state of the cell with the largest payoff in (x, y)'s neighborhood. Any payoffs for cells affected by the possible change in state of cell (x, y) are

[4]Consistent with [80], in Figures B.6 and B.7, blue corresponds to a cooperator cell that was a cooperator in the previous time step, red corresponds to a defector following a defector, green to a cooperator following a defector, and yellow to a defector following a cooperator.



(a) Initial configuration **(b)** Generation $t = 30$

Figure B.6: Synchronous application of the Prisoners' Dilemma

recomputed. Another (x, y) cell is selected at random and the process is repeated. One time step is complete when all cells in the landscape have been selected and updated. Hence, only one copy of the landscape is required. The dilemma evolves asynchronously according to Algorithm B.4.

```
while ( no change in the state of any cell ) {
    do {
        for ($x = 0$; $x < X$; $x$++)
            for ($y = 0$; $y < Y$; $y$++)
                if ( any payoff for $(x, y)$ needs recomputing )
                    SumPayoffs($x$, $y$);

        select an $(x, y)$ cell at random, without replacement;
        EvaluateState($x$, $y$);   /* in sequence using one landscape copy */
    } while ( there remain cells to be selected );
}
```

Algorithm B.4: Prisoners' Dilemma algorithm — asynchronous application

Similar to the results in [51], kaleidoscopic patterns no longer emerge if we update the landscape asynchronously. For the initial configuration given in Figure B.6(a), Figure B.7 shows the landscape after $t = 30$ time steps using asynchronous updating[5]. Notice that, unlike with synchronous updating, there is a majority of defectors and there are no symmetrical patterns. After only a few more time steps, the landscape achieves a stable configuration of all defectors. These results provide further evidence that asynchronous updating may be desirable because synchronous updating can "introduce spurious, undesired symmetries" [105].

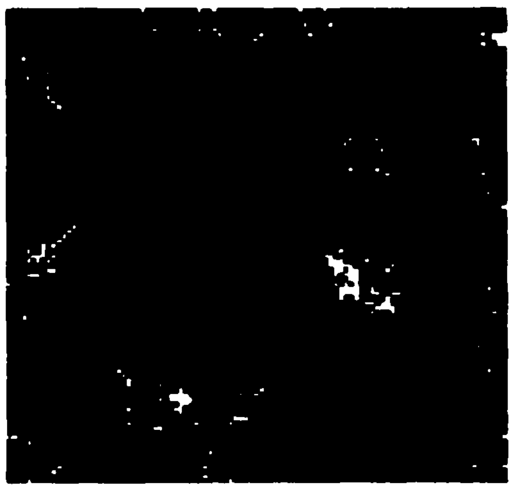


Figure B.7: Asynchronous application of the Prisoners' Dilemma, $t = 30$

[5]In the asynchronous case, the resulting configuration is dependent on the random variate sequence of (x, y) cells selected for updating, as determined by the choice of initial seed. We use 123456789 as the initial seed for the results shown.

Bibliography

[1] Matthew H. Austern. *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*. Addison-Wesley, 1999.

[2] Robert Axelrod. *The Complexity of Cooperation*. Princeton University Press, Princeton, NJ, 1997.

[3] Robert Axelrod and William D. Hamilton. The evolution of cooperation. *Science*, 211:1390–1396, March 1981.

[4] Robert Axtell, Robert Axelrod, Joshua M. Epstein, and Michael D. Cohen. Aligning simulation models: A case study and results. *Computational and Mathematical Organization Theory*, 1:123–141, 1996.

[5] Per Bak and K. Chen. A forest-fire model and some thoughts on turbulence. *Physics Letters A*, 147(5-6):297–299, 1990.

[6] Per Bak and Chao Tang. Earthquakes as a self-organized critical phenomenon. *Journal of Geophysical Research*, 94(B11):15635–15637, November 1989.

[7] Jerry Banks, John S. Carson II, Barry L. Nelson, and David M. Nicol. *Discrete-Event System Simulation*. Prentice Hall, third edition, 2001.

[8] B. Bode, D.M. Halstead, R. Kendall, and Z. Lei. The Portable Batch Scheduler and the Maui scheduler on Linux clusters. In *Proceedings of the 4th Annual Linux Showcase and Conference*, pages 217–224, Atlanta, GA, October 2000.

[9] Eric Bonabeau, Guy Theraulaz, and Jean-Louis Deneubourg. The synchronization of recruitment-based activities of ants. Santa Fe Institute Working Paper 98-01-002, Santa Fe, NM, January 1998.

[10] Randy Brown. Calendar queues: A fast $O(1)$ priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10):1220–1227, October 1988.

[11] C. H. Builder and S. C. Bankes. Artificial Societies: A concept for basic research on the societal impacts of information technology. *RAND Report P-7740*, 1991.

[12] Roger Burkhart. The Swarm multi-agent simulation system. Position paper for OOPSLA '94 Workshop on 'The Object Engine', 1994.

[13] ROGER BURKHART. Schedules of activity in the Swarm simulation system. Position paper for OOPSLA '97 Workshop on OO Behavioral Semantics, 1997.

[14] Arthur W. Burks, editor. *Essays on Cellular Automata.* University of Illinois Press, Urbana, IL, 1970.

[15] S.H. CHIANG, R.K. MANSHARAMANI, AND M.K. VERNON. Use of application characteristics and limited preemption for run-to-completion parallel processor scheduling policies. In *Proceedings of the 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 33–44, 1994.

[16] S.H. CHIANG AND M.K. VERNON. Production job scheduling for parallel shared memory systems. In *Proceedings of the International Parallel and Distributed Processing Symposium 2001 (IPDPS 2001)*, San Francisco, CA, April 2001.

[17] J. CORBALAN AND J. LABARTA. Performance-driven processor allocation. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation (OSDI)*, pages 59-72, San Diego, CA, October 2000.

[18] J. CORBALAN, X. MARTORELL, AND J. LABARTA. Improving gang scheduling through job performance analysis and malleability. In *Proceedings of the International Conference on Supercomputing 2001*, pages 303–311, Sorrento, Italy, 2001.

[19] PAUL DAVIDSSON. Agent based social simulation: A computer science view. *Journal of Artificial Societies and Social Simulation*, 5(1), January 2002.

[20] E. DEELMAN AND B.K. SZYMANSKI. Simulating Lyme disease using parallel discrete event simulation. In *Proceedings of the 1996 Winter Simulation Conference*, J.M. Charnes, D.J. Morrice, D.T. Brunner, and J.J. Swain, editors, pages 46–53, 1996.

[21] E. DEELMAN AND B.K. SZYMANSKI. Breadth-first rollback in spatially explicit simulations. In *Workshop on Parallel and Distributed Simulation (PADS '97)*, pages 124–131. IEEE, 1997.

[22] E. DEELMAN AND B.K. SZYMANSKI. Dynamic load balancing in parallel discrete event simulation for spatially explicit problems. In *Workshop on Parallel and Distributed Simulation (PADS '98)*, pages 46–53. IEEE, 1998.

[23] JORDI DELGADO AND RICHARD V. SOLÉ. Self-synchronization and task fulfillment in social insects. Santa Fe Institute Working Paper 98-08-069, Santa Fe, NM, August 1998.

[24] Jacques Demongeot, Eric Golés, and M. Tchuente, editors. *Dynamical Systems and Cellular Automata.* Academic Press, London, 1985.

[25] CATHERINE DIBBLE. Theory in a complex world: Agent-based simulations of geographic systems. In *First International Conference on GeoComputation*, Leeds, UK, September 1996.

[26] DOUGLAS D. DONALSON AND ROGER M. NISBET. Population dynamics and spatial scale: Effects of system size on population persistence. *Ecology*, 80(8), 1999.

[27] JOSHUA M. EPSTEIN. Zones of cooperation in demographic Prisoner's Dilemma. Santa Fe Institute Working Paper 96-06-042, Santa Fe, NM, December 1997.

[28] JOSHUA M. EPSTEIN AND ROBERT AXTELL. *Growing Artificial Societies.* Brookings Institution Press, Washington, D.C., 1996.

[29] DROR G. FEITELSON. Metrics for parallel job scheduling and their convergence. In *Proceedings of the Seventh Workshop on Job Scheduling Strategies for Parallel Processing*, D. G. Feitelson and L. Rudolph, editors, volume 2221 of *Lecture Notes in Computer Science*, pages 188–206. Springer–Verlag, 2001.

[30] Parallel Workload Archive. `http://www.cs.huji.ac.il/labs/parallel/workload/`.

[31] GEORGE S. FISHMAN. *Principles of Discrete Event Simulation.* Wiley-Interscience, 1978.

[32] WILLIAM R. FRANTA AND KURT MALY. A comparison of heaps and the TL structure for the simulation event set. *Communications of the ACM*, 21(10):873–875, October 1978.

[33] ROSS A. GAGLIANO AND MICHAEL R. LAUER. Discrete element models and real life duals. In *Proceedings of the 1994 Winter Simulation Conference*, J.D. Tew, S. Manivannan, D.A. Sadowski, and A.F. Seila, editors, pages 625–632, 1994.

[34] MARTIN GARDNER. The fantastic combinations of John Conway's new solitaire game 'Life'. *Scientific American*, 223(4):120–123, April 1970.

[35] D. GHOSAL, G. SERAZZI, AND S.K. TRIPATHI. Processor working set and its use in scheduling multiprocessor systems. *IEEE Transactions on Software Engineering*, 17(5):443–453, May 1991.

[36] NIGEL GILBERT. Simulation: An emergent perspective. Conference on New Technologies in the Social Sciences, LAFORIA (Paris), October 1995.

[37] Nigel Gilbert and Rosaria Conte, editors. *Artificial Societies: The Computer Simulation of Social Life.* UCL Press, London, 1995.

[38] Nigel Gilbert and Jim Doran, editors. *Simulating Societies: The Computer Simulation of Social Phenomena.* UCL Press, London, 1994.

[39] NIGEL GILBERT AND KLAUS TROITZSCH. *Simulation for the Social Scientist.* Open University Press, Buckingham, 1999.

[40] K. GLASS, M. LIVINGSTON, AND J. CONERY. Distributed simulation of spatially explicit ecological models. In *Workshop on Parallel and Distributed Simulation (PADS '97)*, pages 60–63. IEEE, 1997.

[41] HARVEY GOULD AND JAN TOBOCHNIK. *An Introduction to Computer Simulation Methods: Applications to Physical Systems (Part 2).* Addison-Wesley, Reading, MA, 1988.

[42] G. GRINSTEIN AND C. JAYAPRAKASH. Simple models of self-organized criticality. *Computers in Physics*, 9(2):164, 1995.

[43] Howard Gutowitz, editor. *Cellular Automata: Theory and Experiment.* MIT Press, Cambridge, MA, 1991.

[44] DIANE HARROLD. Economic markets within an artificial society model. Technical report, Dept. of Computer Science, College of William & Mary, May 2000.

[45] RAINER HEGSELMANN. Cellular automata in the social sciences: Perspectives, restrictions and artefacts. *Modelling and Simulation in the Social Sciences from the Philosophy of Science Point of View*, pages 209–233, 1996.

[46] RAINER HEGSELMANN AND ANDREAS FLACHE. Understanding complex social dynamics: A plea for cellular automata based modelling. *Journal of Artificial Societies and Social Simulation*, 1(3), June 1998.

[47] JAMES O. HENRIKSEN. An improved events list algorithm. In *Proceedings of the 1977 Winter Simulation Conference*, R. Sargent, J.W. Schmidt, and H.J. Highland, editors, pages 547–557, 1977.

[48] JAMES O. HENRIKSEN. Event list management — a tutorial. In *Proceedings of the 1983 Winter Simulation Conference*, J. Banks, B. Schmeiser, and S. Roberts, editors, pages 542–551, 1983.

[49] JOHN HOLLAND. *Hidden Order: How Adaptation Builds Complexity.* Addison-Wesley, 1995.

[50] ELLIS HOROWITZ, SARTAJ SAHNI, AND DINESH MEHTA. *Fundamentals of Data Structures in C++.* Computer Science Press, 1995.

[51] BERNARDO A. HUBERMAN AND NATALIE S. GLANCE. Evolutionary games and computer simulations. *Proceedings of the National Academy of Sciences*, 90:7716–7718, August 1993.

[52] IBM LoadLeveler.
`http://www-1.ibm.com/servers/eserver/pseries/software/sp/loadleveler.html`.

[53] YANNIS M. IOANNIDES. Evolution of trading structures. Santa Fe Institute Working Paper 96-04-020, Santa Fe, NM, April 1996.

[54] *The Journal of Artificial Societies and Social Simulation.* Nigel Gilbert, editor. `http://jasss.soc.surrey.ac.uk/`.

[55] ARNE JONASSEN AND OLE-JOHAN DAHL. Analysis of an algorithm for priority queue administration. *BIT Numerical Mathematics*, 15(4):409–422, April 1975.

[56] DOUGLAS W. JONES. An empirical comparison of priority-queue and event-set implementations. *Communications of the ACM*, 29(4):300–311, April 1986.

[57] CATHOLIJN M. JONKER AND JAN TREUR. Agent-based simulation of animal behaviour. CWI National Research Institute for Mathematics and Computer Science, December 1998. Report SEN-R9835.

[58] P. KELEHER, D. ZOTKIN, AND D. PERKOVIC. Attacking the bottlenecks in backfilling schedulers. *Cluster Computing: The Journal of Networks, Software Tools and Applications*, 3(4):245–254, 2000.

[59] JEFFREY H. KINGSTON. Analysis of Henriksen's algorithm for the simulation event set. *SIAM Journal of Computing*, 15(3):887–902, August 1986.

[60] Christopher G. Langton, editor. *Artificial Life: An Overview.* MIT Press, Cambridge, MA, 1997.

[61] AVERILL M. LAW AND W. DAVID KELTON. *Simulation Modeling and Analysis.* McGraw-Hill, third edition, 1999.

[62] BARRY G. LAWSON AND STEVE PARK. Asynchronous time evolution in an artificial society model. *Journal of Artificial Societies and Social Simulation*, 3(1), January 2000.

[63] BARRY G. LAWSON AND EVGENIA SMIRNI. Multiple-queue backfilling scheduling with priorities and reservations for parallel systems. In *Proceedings of the Eighth Workshop on Job Scheduling Strategies for Parallel Processing* (to appear), Edinburgh, Scotland, July 2002.

[64] BARRY G. LAWSON, EVGENIA SMIRNI, AND DANIELA PUIU. Self-adapting backfilling scheduling for parallel systems. In *International Conference on Parallel Processing* (to appear), Vancouver, B.C., August 2002.

[65] P. Maes, editor. *Designing Autonomous Agents: Theory and Practice from Biology to Engineering and Back.* MIT Press, Cambridge, MA, 1990.

[66] STEVE MAINS. *Optimizing Combat Capabilities by Modeling Combat as a Complex Adaptive System.* PhD thesis, Dept. of Computer Science, College of William & Mary, 2002. To Appear.

[67] MAURICIO MARÍN. An empirical comparison of priority queue algorithms. Technical Report PRG-TR-10-97, University of Oxford, 1997.

[68] J.P. MARNEY AND HEATHER TARBERT. Why do simulation? Towards a working epistemology for practitioners of the dark arts. *Journal of Artificial Societies and Social Simulation*, 3(4), October 2000.

[69] Maui Scheduler Open Cluster Software. `http://mauischeduler.sourceforge.net/`.

[70] C. McCann, R. Vaswani, and J. Zahorjan. A dynamic processor allocation policy for multiprogrammed shared memory multiprocessors. *ACM Transactions on Computer Systems*, 11(2):146–178, May 1993.

[71] William M. McCormack and Robert G. Sargent. Analysis of future event-set algorithms for discrete event simulation. *Communications of the ACM*, 24(12):801–812, December 1981.

[72] Barry McMullin. SCL: An artificial chemistry in Swarm. Santa Fe Institute Working Paper 97-01-002, Santa Fe, NM, January 1997.

[73] Nelson Minar, Roger Burkhart, Chris Langton, and Manor Askenazi. The Swarm simulation system: A toolkit for building multi-agent simulations. Santa Fe Institute Working Paper 96-06-042, Santa Fe, NM, June 1996. `http://www.swarm.org/`.

[74] Michael Möhring. Social science multilevel simulation with MIMOSE. In *Social Science Microsimulation*, K.G. Troitsch, U. Müller, N.E. Gilbert, and J. Doran, editors, pages 123–137. Springer–Verlag, 1996.

[75] Nathan T. Moore. Artificial Societies: A computational model of disease transmission. Master's thesis, Dept. of Computer Science, College of William & Mary, August 1999.

[76] Rachel I. Moore. Artificial Societies: A computational approach to studying combat. Master's thesis, Dept. of Computer Science, College of William & Mary, July 1999.

[77] Scott Moss, Helen Gaylard, Steve Wallis, and Bruce Edmonds. SDML: A multi-agent language for organizational modelling. CPM Report 97-16, Centre for Policy Modelling, Manchester Metropolitan University, March 1997.

[78] A. Mualem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6):529–543, June 2001.

[79] H. J. Müller, Th. Malsch, and I. Shulz-Schaeffer. Socionics: Introduction and potential. *Journal of Artificial Societies and Social Simulation*, 1(3), June 1998.

[80] Martin A. Nowak and Robert M. May. Evolutionary games and spatial chaos. *Nature*, 359:826–829, October 1992.

[81] J. Ousterhout. Scheduling techniques for concurrent systems. In *Proceedings of the Third International Conference on Distributed Computing System*, pages 22–30, October 1982.

[82] Steve Park and Lawrence Leemis. *Discrete-Event Simulation: A First Course*. College of William and Mary, Williamsburg, VA, 1999.

[83] MILES T. PARKER. What is Ascape and why should you care? *Journal of Artificial Societies and Social Simulation*, 4(1), January 2001.

[84] E.W. PARSONS AND K.C. SEVCIK. Coordinated allocation of memory and processors in multiprocessors. In *Proceedings of the 1996 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 57–67, May 1996.

[85] Portable Batch System. `http://www.openpbs.org/`.

[86] D. PERKOVIC AND P. KELEHER. Randomization, speculation, and adaptation in batch schedulers. In *Proceedings of Supercomputing 2000 (SC2000)*, November 2000.

[87] A. REPENNING, A. IOANNIDOU, AND J. ZOLA. AgentSheets: End-user programmable simulations. *Journal of Artificial Societies and Social Simulation*, 3(3), June 2000.

[88] MITCHELL RESNICK. *Turtles, Termites and Traffic Jams: Explorations in Massively Parallel Microworlds (Complex Adaptive Systems)*. MIT Press, Cambridge, MA, 1997.

[89] C. REWERTS, P. SYDELKO, J. DOLPH, A. SHAPIRO, AND T. TAXON. An object-oriented, individual-based, spatially explicit environmental model. In *4th International Conference on Integrating GIS and Environmental Modeling (GIS/EM4)*, Alberta, Canada, September 2000.

[90] ALMA RISKA, WEI SUN, EVGENIA SMIRNI, AND GIANFRANCO CIARDO. AdaptLoad: effective balancing in clustered web servers under transient load conditions. In *International Conference on Distributed Computing Systems (ICDCS 2002)* (to appear), Vienna, Austria, July 2002.

[91] ROBERT RÖNNGREN AND RASSUL AYANI. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation*, 7(2):157–209, April 1997.

[92] ROBERT RÖNNGREN, JENS RIBOE, AND RASSUL AYANI. Lazy queue: A new approach to implementing the pending event set. *International Journal in Computer Simulation*, 3:303–332, 1993.

[93] E. ROSTI, E. SMIRNI, L.W. DOWDY, AND G. SERAZZI. Robust partitioning policies for multiprocessor systems. *Performance Evaluation*, 19(2-3):141–165, 1994.

[94] E. ROSTI, E. SMIRNI, G. SERAZZI, L.W. DOWDY, AND K.C. SEVCIK. Processor saving scheduling policies for multiprocessor systems. *IEEE Transactions on Computers*, 47(2):178–189, February 1998.

[95] THOMAS SAUERBIER. UMDBS - a new tool for dynamic microsimulation. *Journal of Artificial Societies and Social Simulation*, 5(2), March 2002.

[96] THOMAS SCHELLING. *Micromotives and Macrobehavior*. Norton, 1978.

[97] B. SCHROEDER AND M. HARCHOL-BALTER. Evaluation of task assignment policies for supercomputing servers: The case for load unbalancing and fairness. In *Proceedings of the 9th IEEE Symposium on High Performance Distributed Computing (HPDC '00)*, pages 211–220, August 2000.

[98] ROBERT SEDGEWICK. *Algorithms in C++*. Addison–Wesley, 3rd edition, 1998.

[99] S. SETIA, M.S. SQUILLANTE, AND S.K. TRIPATHI. Analysis of processor allocation in multiprogrammed parallel processing systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(4):401–420, April 1994.

[100] DANIEL SLEATOR AND ROBERT TARJAN. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, July 1985.

[101] E. SMIRNI, E. ROSTI, L.W. DOWDY, AND G. SERAZZI. Adaptive scheduling policies for multiprocessor systems. *Journal of Systems Architecture*, 44(9):703–721, June 1998.

[102] D. TALBY AND D.G. FEITELSON. Supporting priorities and improving utilization of the IBM SP2 scheduler using slack-based backfilling. In *Proceedings of the 13th International Parallel Processing Symposium*, pages 513–517, April 1999.

[103] PIETRO TERNA. Simulation tools for social scientists: Building agent based models with Swarm. *Journal of Artificial Societies and Social Simulation*, 1(2), March 1998.

[104] G. THEODOROPOULOS AND B. LOGAN. A framework for the distributed simulation of agent-based systems. In *Proceedings of the 13th European Simulation Multiconference (ESM '99)*, pages 58–65. Society for Computer Simulation, 1999.

[105] TOMASO TOFFOLI AND NORMAN MARGOLUS. *Cellular Automata Machines: A New Environment for Modeling*. MIT Press, Cambridge, MA, 1987.

[106] A. TUCKER AND A. GUPTA. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pages 159–166, 1989.

[107] JOHN VON NEUMANN. *Theory of Self-Reproducing Automata*. University of Illinois Press, Urbana, IL, 1966. Edited by Arthur W. Burks.

[108] A. MARTIN WILDBERGER. Introduction & overview of 'artificial life' – evolving intelligent agents for modeling & simulation. In *Proceedings of the 1996 Winter Simulation Conference*, J.M. Charnes, D.J. Morrice, D.T. Brunner, and J.J. Swain, editors, pages 161–168, 1996.

[109] GREGORY V. WILSON. From Active X to cargo-cult science. *Dr. Dobb's Journal*, 359:117–119, September 1997.

[110] STEPHEN WOLFRAM. Cellular automata as models of complexity. *Nature*, 311(4):419–424, October 1984.

[111] STEPHEN WOLFRAM. *Theory and Applications of Cellular Automata*. World Scientific Publishing Co. Pte. Ltd., Singapore, 1986.

[112] JEFFREY R. YOUNG. Using computer models to study the complexities of human society. *The Chronicle of Higher Education*, July 1998.

VITA

Barry Glenn Lawson

Barry Glenn Lawson was born in Norton, Virginia on November 29, 1971. He graduated from Coeburn High School in Coeburn, Virginia in 1989. He graduated from Clinch Valley College (now the University of Virginia's College at Wise) in 1993 with a Bachelor of Science degree, majoring in both Mathematics and Computer Information Systems. He entered the College of William and Mary in 1994, received a Master of Science degree in Computer Science in 1996, and will receive a Doctor of Philosophy degree in Computer Science in August 2002. Starting in August 2002, the author assumes a tenure-track position as Assistant Professor in the Department of Mathematics and Computer Science at the University of Richmond in Richmond, Virginia.