BISHOP: A Framework for Economic Simulation Using Cellular Automata

by

Harris L. Gilliam

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 24, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

BISHOP is a system for economic simulation that uses generalized cellular automata as the framework. Agents in the economy are represented by individual cells on a lattice. The user is required to provide procedures that implement a model of agent behavior. The code runs on the CM-2, a massively parallel computer manufactured by Thinking Machines Corporation, and is written in C* a data-parallel extension of the C language. The code consists of a text command interface and *skeletal* code that implements the cellular automata. This thesis describes the motivation behind the creation of BISHOP, the types of economies that can be simulated using BISHOP, issues of data-parallel programming on the Connection Machine CM-2 computer, preliminary results from simulating simple economies and a discussion of the future direction of the project.

Thesis Supervisor: Bruce M. Boghosian
Title: Principal Research Scientist

BISHOP: A Framework for Economic Simulation Using Cellular Automata

by

Harris L. Gilliam

Submitted to the Department of Electrical Engineering and Computer Science
on Aug 24, 1993, in partial fulfillment of the
requirements for the degree of
Bachelor of Science in Computer Science and Engineering

Abstract

BISHOP is a system for economic simulation that uses generalized cellular automata as the framework. Agents in the economy are represented by individual cells on a lattice. The user is required to provide procedures that implement a model of agent behavior. The code runs on the CM-2, a massively parallel computer manufactured by Thinking Machines Corporation, and is written in C* a data-parallel extension of the C language. The code consists of a text command interface and *skeletal* code that implements the cellular automata. This thesis describes the motivation behind the creation of BISHOP, the types of economies that can be simulated using BISHOP, issues of data-parallel programming on the Connection Machine CM-2 computer, preliminary results from simulating simple economies and a discussion of the future direction of the project.

Thesis Supervisor: Bruce M. Boghosian
Title: Principal Research Scientist

Contents

List of Figures

List of Tables

Acknowledgements

I would like to take the time to thank the many people who helped, served and laid down their lives for me. I could not have survived without you !!

First I want to thank my supervisor, Bruce Boghosian, for his patience and support during a truly difficult time.

Many thanks to Lisa Chou, Jude Federspiel, Piper Keebles, Bruce Lewis, Deborah Flynn, Kristi Jenson, Sajjan and everyone else who devoted many precious hours to proofreading this thesis.

Most of all I must thank my God and Savior Jesus for providing me with unending strength, a heart of perseverance and a true understanding of what is real and valuable in life. I can be joyfull... always!

```
Of making many books there is no end,
  and much study wearies the body.

Now all has been heard;
  here is the conclusion of the matter:
Fear God and keep his commandments,
  for this is the whole of man.
For God will bring every deed into
  judgement, including every hidden
  thing, whether it is good or evil.

                Ecclesiastes 12:12-14
```

Chapter 1

Introduction

1.1 Background

Today's scientists and engineers are faced with ever increasing difficulties when trying to simulate complex systems. More often than not a researcher cannot afford to invest the time and resources to construct and directly study a system and many times it is simply impossible to do so. This is particularly true for economics. Economists cannot perturb real economies and watch the results nor can they create or isolate an economy to study. Simulation allows economists to study "hypothetical economies." Simulation is necessary because if we are to really examine large interacting, economic systems, then analytical solutions are no longer possible. It is hoped that the qualitative results of simulations may give insights to build more abstract, analytically feasible models which can demonstrate the same types of qualitative behavior, but this not need to be our only motivation. If we can show that any qualitative behavior is generic in the fundamentals of our model, ex: chaos, then we have showed something already very significant. Very little analysis has been done that focuses on individual behavior, at such a detailed level, that influences the dynamics of economies.

BISHOP is a joint effort by Dio Tsai, an economist, and myself to create a new system for simulating individual behavior using cellular automata (CA). Largely, previous analysis has not attempted to look at dynamic behavior from this level of analysis. All previous work has usually assumed some sort of market process. We have not assumed any such process. What we attempt to examine is the behavioral dynamics of an economy from the fundamentals of individuals that engage in the exchange of goods or services for a price (called agents). Previous attempts, described in price adjustment literature, have always assumed the way agents interact in what is called a "market." We do not define anywhere in our analysis the notion of a market, instead markets arise from the dynamics of our model endogenously. The only similar works are that by Albin and Foley, [1] and Butters [3]. A major task is to find a way of modeling complex dynamical systems that allows easy implementation on current computer architectures without sacrificing too much reality. We will use a class of dynamical systems called cellular automata for economic modeling. In a cellular automaton "cells" are arranged in a regular lattice and have discrete attributes that make up their "state." There is a concept of a "neighborhood" for each cell which consists of some finite number of surrounding cells. Each cell's state is updated based on information obtained from its neighborhood and its current state. This update occurs simultaneously for every cell during each time period. Cells are located discretely in space and time.

We chose CA because out of all the frameworks for possible analysis, to a first approximation, CA are a very natural model of individual autonomous agents. In fact, by having a very loose interpretation of CA, we see that it is exactly isomorphic to a set of agents who interact somewhat locally and have a common time frame in which to base their actions.

1.2 Objective and Scope

Cellular automata have in recent years be successfully used to model complex dynamic systems. The cellular automata model is a simple one, which makes it very attractive, yet it provides a surprisingly powerful means of representing dynamic systems. CA have proven to be very useful in studying complex systems of relatively simple locally interacting units. The simulation of economic agents lends itself to the CA model. Our goal is to provide the economist with a new tool for investigating economic behavior.

The economy is modeled as a three-dimensional lattice of agents who set prices for goods they desire to sell and upper limits on what they are willing to pay (cutoffs) for goods they wish to buy. We chose a three-dimensional lattice to provide more connectivity between agents and because it is easier to visualize complex behavior in three dimensions. These agents must make these decisions based solely on information obtained from their neighborhood. BISHOP provides this CA framework and the user is responsible for developing a model that determines how agents behave and how information travels throughout the lattice.

1.3 Approach

I chose a parallel architecture for the simulation because of the inherently parallel nature of CA. One of the defining features of CA is that the state of the 'cells' is updated in parallel. Most implementations of CA systems have been done on serial computers that simulate this concurrent state updating. Toffoli and Margolus have built dedicated Cellular Automata Machines (CAM) [7] that operate on the state space in parallel. I needed more computational power than was available with the CAM machines and still wanted to implement the simulation in parallel.

The Connection Machine [1] CM-2 computer is a massively parallel Single Instruction Multiple Data (SIMD) architecture and is well suited for this type of application. I chose the CM-2 computer because of its availability (there are two CM-2 computers at MIT and one CM-2 computer is open to public use at Thinking Machines Corporation) and its previous use with CA models. [2] The program is written in C*, an extension to ANSI C designed for data-parallel programming. C* was chosen (CM Fortran and *LISP were also available) [2] because of my extensive experience with C and the UNIX [3] programming environment. The SIMD architecture involves many processors that execute the same instructions on different pieces of data. Each processor in the CM-2 computer has its own bit-addressable memory and is interconnected in an 11 dimensional hypercube configuration. Processors can communicate with each other and pass data. The CM-2 computer supports the abstraction of *virtual processors*, so that each physical processor may be viewed as many virtual processors. The CM-2 computer at Thinking Machines Corporation has 32768 processors. The CM-2 computer at the MIT Artificial Intelligence Laboratory (AI Lab) has 8192 processors. Each cell in the automata lattice is represented by a virtual processor. The lattice has three dimensions (of equal length). Two lattice sizes were studied: a 32^3 lattice with 32768 agents on the CM-2 computer at Thinking Machines, and a 16^3 lattice with 4096 agents on the CM-2 computer at the AI Lab.

The program is implemented in two parts: The command interface and the CA skeleton.

---

[1] Connection Machine is a registered trademark of Thinking Machines Corporation

[2] C*, CM Fortran and *Lisp are all registered trademarks of Thinking Machines Corporation

[3] UNIX is a registered trademark of USL

The command interface provides a friendly interface to the system. It gives the user a way to specify options. It allows the user to set up statistics files where values can be selectively written from the CA state space. It may be extended by the user to provide extra flexibility or features as needed. The skeleton enforces the CA framework. It controls the sequence of events that make up each period of activity in the economy. The skeleton accomplishes this through a multistage process using procedures provided by the user. These procedures implement the details of the model being studied.

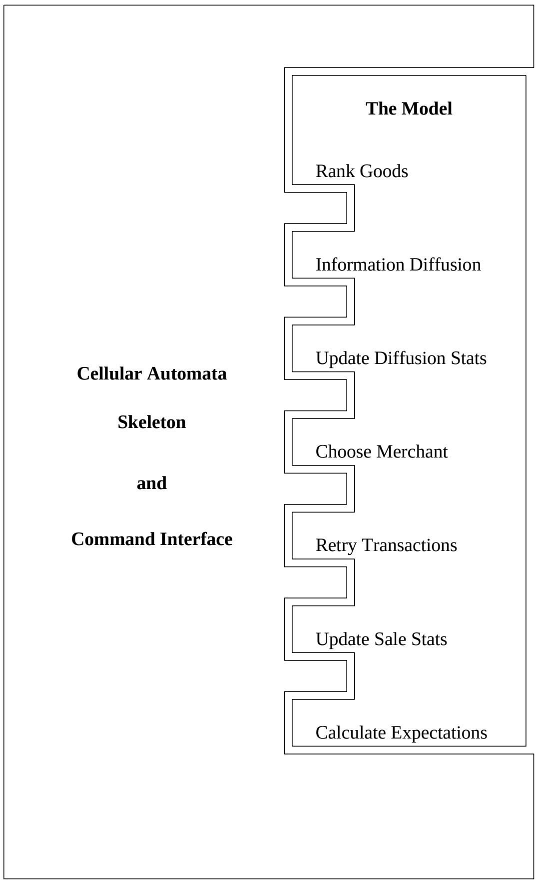

Figure 1-1: Program Pieces

The model is to be coded by the user from the view point of one agent participating in the economy. The skeleton uses procedures provided by the user to complete the seven stages of each period of the simulation.

Ranking Goods: The agent is given the opportunity to rank the goods that are available and decide which goods are to be bought and which are to be sold based on that ranking. Prices are fixed for goods being sold and cutoffs for good being bought. This stage is optional.

Information Diffusion: The agent is allowed to propagate information about prices for goods marked to be sold (ads) into the surrounding neighborhood. Typically information travels by means of a diffusion process. This stage is optional.

Updating Diffusion Statistics: The agent is given a chance to process the newly obtained information from the diffusion stage and collect statistics about it. This stage is optional.

Choosing a Merchant: The agent that plans to purchase a good is required to choose a merchant to buy from. The choice may be based on information obtained during the diffusion stage, derived information or accumulated information. This is the only stage that is not optional.

Retrying Transactions: After choices have been made, the skeleton attempts to enact each agent's plans. The skeleton checks to see that the agents have valid plans and carries them out if possible. The state space is updated and a record is kept of which transactions were successful and which failed and why. This information is available to the agent to be used in future decisions. When an agent's plan fails to be enacted the agent is given a chance to retry the transaction by creating a new plan. The details of retrying a transaction are the responsibility of the user. Retrying a failed transaction is optional.

Update Sale Statistics: The agent is given the chance to collect statistics about the success or failure of the transactions attempted in the previous stage. This stage is optional.

Calculating Expectations: The agent is given the chance to calculate and/or change expectations and beliefs about the market. Typically prices for goods being sold and cutoffs for goods being bought are adjusted here. This stage is optional.

1.4 Outline

The bulk of this thesis focuses on data-parallel algorithms used to implement the CA skeleton and two models of agent behavior. Problems for which well know solutions exist for serial architectures are addressed in light of the data-parallel programming environment of the CM-2 computer. A short discussion of preliminary data collected from a naive model of agent behavior is included. The rest of the thesis is organized as follows:

- Chapter 2 describes various aspects of economic modeling that can be explored using BISHOP and defines expectations for what can be learned from these types of simulations.

- Chapter 3 gives a formal definition of cellular automata and the differences from classical CA and our model.

- Chapter 4 describes the data-parallel programming environment of the CM-2 computer. It gives a brief explanation of the extensions to the C programming language that facilitate the coding of parallel algorithms. Finally it defines the basic concepts of the Connection Machine computer programming methodology. C* code fragments are present and it is assumed that the reader has at least basic proficiency with C.

- Chapter 5 describes the CI to BISHOP. The CI provides a *user friendly* environment for economic simulation. It is extensible by the user and the facilities currently provided are explained as well as the means for extending its functionality. As in chapter 4 it is assumed the reader has basic proficiency with C.

- Chapter 6 describes how the CA skeleton works and the major algorithmic issues involved in its coding.

- Chapter 7 describes the basic problems faced when coding models of economic behavior. It outlines the simplifying assumptions made for the two models presented and what impact these assumptions had on the code. Finally it gives a detailed explanation of the major algorithms realized by the code.

- Chapter 8 gives a general overview of how to run BISHOP and collect data for future analysis.

- Chapter 9 presents preliminary data collected using BISHOP and a short discussion of its significance.

- Appendix A gives a general flowgraph of the BISHOP code outlining how the major modules interact. Also detailed specifications are given for each of the major procedures and data structures used.

- Appendix B includes a full listing of the BISHOP source code.

- Appendix C defines the various mathematical symbols used throughout the thesis as well as typesetting conventions.

Chapter 2

Economic Modeling

BISHOP places few constraints on the types of economies that can be modeled. The only thing that is required of each model is that agents have a way of choosing who to buy goods from. Very complex or very simple economies may be studied using this open ended approach. Economic systems, such as exchange economies, can be very easily understood using CA. CA simulations can for the first time ever provide the economist with a tool for looking at dynamic behavior from a disaggregated point of view. There are many classical hypothesis in economics that may be confirmed or refuted in this dynamic framework. One such hypothesis is the development of an equilibrium. There may be many notions of equilibrium, for example:

- In the sense of having the price asked for a good equal to what agents are willing to pay.
- In the sense that demands or sales that "reproduce" themselves each period.
- In the sense that all trading stops.

We can attempt to look at all of these situations. There are also questions concerning the time it takes to reach equilibrium and whether the path to equilibrium cause a drastic change in the welfare or wealth of consumers. All of this is related to questions about growth and development in countries. These are but a beginning to the types of questions that may be answered.

2.1 Models of Agent Behavior

Initially two models of agent behavior were to be implemented and studied. One model, hedonistic dain bramaged, was primarily designed to test the CA skeleton code. It is to naive to be considered a serious economic model but still proved to provide interesting data. The other, marginal utility ranking with adaptive price/expectation setting, was designed as a valid model of economic behavior. The second model will be described in detail, but its complete implementation is outside the scope of this thesis.

2.2 The Pure Exchange Economy

Both models described in this section are designed to work in a pure exchange economy. A pure exchange economy is an economy with no production or consumption. Agents are often infinitely lived, as in these models. Sometimes there is a medium of exchange that serves as money. These models have a Cash-In-Advance constraint where agents must possess the cash reserves to purchase goods at the time of the transaction, hence there is no notion of credit. The pure exchange economy is usually characterized by the set of agents, the set of goods, the preferences of the agents over the set of goods, and perhaps a money commodity.

2.3 Slow Diffusion and Market Omnipotence

The two models of information propagation examined are slow diffusion and market omnipotence. With slow diffusion information about agents' planned sales is released via a stochastic process. Each agent that is selling a good distributes its price information to each location in the immediate neighborhood with a fixed probability. Agents also distribute any price information they have accumulated. This model of diffusion was described in Toffoli and Margolus's book [7, pp. 155-159]. It involves only local interactions and is shown to behave like a diffusion process over long time periods. When an agent changes his price, the old price instantly disappears from the registers of all agents. This imposes a cost for changing prices, agents that do not change their prices allow them to diffuse farther, possibly opening up new markets. On the other hand, the uncertainty of the existence of these new markets is an incentive for agents to change their prices with the hopes that a new price will bring immediate profits. Much economic literature is devoted to how economic agents solve this problem. The alternative to slow diffusion is market omnipotence. In this model information about prices is available to all agents everywhere in the lattice instantly. There is also much economic literature devoted to what happens under this "perfect" information situation.

2.4 Utility and Marginal Utility

Utility is defined as a measure of the "internal value" of a specific good to each agent, i.e. the agent has some measure for each possible bundle of a good (inventory) that he might hold. Utility is a function of inventory and it is assumed to be nondecreasing and quasi-concave.[8] Utility ($U()$) can never be negative and must increase with increasing inventory.

$$I_0 < I_1 \Longleftrightarrow U(I_0) < U(I_1)$$

The first and second derivatives of $U()$ must always be less than or equal to zero, this ensures that $U()$ is quasi-concave.

$$U' \geq 0$$

$$U'' \leq 0$$

It is possible to derive this "utility function" from a set of preferences that satisfy the above assumptions. Marginal Utility is defined as the change in utility due to the incremental change in the inventory of a good [8]. Given functions for utility and inventory marginal utility can be found as follows:

$$MU \stackrel{\text{def}}{=} \frac{\partial U}{\partial X}$$

2.5 Adaptive Price/Expectation Setting

An important aspect of agent behavior is the formation and updating of expectations and beliefs about the market. Agents maintain an expectation of the market value of goods in the economy as well as a belief about the optimal price at which they can sell their goods. Adaptive behavior was chosen to update this expectation and belief because it is conceptually simple as a description of agent behavior and is easy to implement. At the end of each period a measure of success in selling goods is computed. We define R^{*t}_{ij} to be the units of good j sold by agent i during period t and R^{t}_{ij} to be the units of good j that agent i planned to sell during period t. The measure of success/failure is $\frac{R^{*t}_{ij}-R^{t}_{ij}}{R^{t}_{ij}}$, where negative values represent failure to sell the planned units. The optimal price for the next period is calculated from the previous price, the measure of success/failure and a weight (λ_{ij}):

$$P^{*t+1}_{ij} = P^{*t}_{ij}(1 + \lambda_{ij}\frac{R^{*t}_{ij} - R^{t}_{ij}}{R^{t}_{ij}}) \tag{2.1}$$

We define $\tilde{P}^{t}_{ij}$ as the lowest price gathered from agents i's neighborhood during period t for good j. The expected price for the next period is:

$$E^{t+1}_{i}[p_j] = E^{t}_{i}[p_j] + \lambda_{ij}(\tilde{P}^{t}_{ij} - E^{t}_{i}[p_j]) \tag{2.2}$$

$0 < \lambda_{ij} < 1$

2.6 Hedonistic Dain Bramaged

These agents are called hedonistic because they plan to satisfy their desires each period regardless of the economic feasibility of that plan. They are called dain bramaged (a contortion of brain damaged) because they aren't very smart about how they formulate expectations and beliefs. This model is designed to test the BISHOP code and is not meant to represent any valid model of economic behavior. The results obtained are nonetheless interesting. This model of agent behavior reflects irrationality in the decision making process. Agents are born with a desire, Δ_{ij}, for each good j that does not change throughout their lifetime. This desire determines the quantity of goods to be bought or sold. During each period agents attempt to buy or sell the desired quantity of goods. Initially each agent's disposition towards each good and its price/cutoff is randomly chosen. No agent will offer a good for sale below 10 price units. They maintain a cumulative record of the highest $({}^{seen}P^{t}_{ij})$ and lowest $({}_{seen}P^{t}_{ij})$ prices they have gathered from their neighborhood for each good and the highest $({}^{paid}P^{t}_{ij})$ lowest $({}_{paid}P^{t}_{ij})$ prices paid for each good (either by the agent to a merchant or by a customer to the agent). These are initialized to a default value before the simulation begins:

$$_{paid}P^t_{ij} =_{seen} P^t_{ij} = \infty$$

$$^{paid}P^t_{ij} =^{seen} P^t_{ij} = 0.0$$

This model only provides actions for four of the seven simulation stages. It applies a simple heuristic during the "calculate expectations" stage that ensures that prices, money and inventory are constantly changing over time. This model is designed to test the code and it would be undesirable for an equilibrium to arise. This model is designed to run using an economy where agents have access to all the prices offered by every merchant in the economy. This is the market omnipotence situation. The actions at various stages are described below:

Rank Goods: No action defined.

Information Diffusion: No action defined.

Update Diffusion Statistics: Let ap^H_j and ap^L_j be the highest and lowest advertised prices seen for good j respectively:

$$^{seen}P^t_{ij} = \max[ap^H_j,^{seen} P^t_{ij}] \tag{2.3}$$

$$_{seen}P^t_{ij} = \min[ap^L_j,_{seen} P^t_{ij}] \tag{2.4}$$

Choose Merchant: Choose the merchant with the lowest price below cutoff ($E^t_i[p_j]$).

Retry Transaction: No action defined.

Update Sale Statistics: Let p^H_j and p^L_j be the highest and lowest prices paid for good j, if the agent is a customer of good j, or received for good j, if the agent is a merchant of good j, respectively:

$$^{paid}P^t_{ij} = \max[p^H_j,^{paid} P^t_{ij}] \tag{2.5}$$

$$_{paid}P^t_{ij} = \min[p^L_j,_{paid} P^t_{ij}] \tag{2.6}$$

Calculate Expectations: Let Q_{ij}^{*t} be the number of units of good j bought by agent i during period t.

$$\forall j, R_{ij}^t > 0, R_{ij}^{*t} = 0, P_{ij}^{*t} > 11.11 \quad . \quad P_{ij}^{*t+1} = (.9)E_i^t[p_j] \tag{2.7}$$

$$\forall j, R_{ij}^t > 0, R_{ij}^{*t} = 0, P_{ij}^{*t} \leq 11.11 \quad . \quad \begin{pmatrix} R_{ij}^{t+1} = 0 \\ Q_{ij}^{t+1} = \Delta_{ij} \\ P_{ij}^{*t+1} = \min\left[\frac{{}_{paid}P_{ij}^t + (1.1)^{paid}P_{ij}^t}{2}, 50.0\right] \end{pmatrix} \tag{2.8}$$

$$\forall j, R_{ij}^t > 0, X_{ij}^t > 0 \quad . \quad P_{ij}^{*t+1} = (1.1)P_{ij}^{*t} \tag{2.9}$$

$$\forall j, R_{ij}^t > 0, X_{ij}^t = 0 \quad . \quad \begin{pmatrix} R_{ij}^{t+1} = 0 \\ Q_{ij}^{t+1} = \Delta_{ij} \\ P_{ij}^{*t+1} = \frac{{}_{paid}P_{ij}^t + (1.1)^{paid}P_{ij}^t}{2} \end{pmatrix} \tag{2.10}$$

$$\forall j, Q_{ij}^t > 0, Q_{ij}^{*t} = 0 \quad . \quad \begin{pmatrix} Q_{ij}^{t+1} = 0 \\ R_{ij}^{t+1} = \min\left[X_{ij}^t, \Delta_{ij}\right] \\ P_{ij}^{*t+1} = \min\left[\frac{{}_{paid}P_{ij}^t + (1.1)^{paid}P_{ij}^t}{2}, 50.0\right] \end{pmatrix} \tag{2.11}$$

2.7 Marginal Utility Ranking

This model uses the concept of marginal utility to decide which goods are to be bought and sold. Goods are ranked by increasing value of their marginal utility: $\frac{\eta_{ij}}{(E_i^t[p_j])(X_{ij}^t)}$, where η_{ij} is randomly chosen for each agent, taking a value between zero and one, for each good j. X_{ij}^t is agent i's inventory, at time t, of good j. This ranking is partitioned into goods that are sold and goods that are bought. The goods at the lower end of the ranking are more likely to be sold while those at the upper end are likely to be bought. The number of goods to be bought or sold ($\vec{Q}_i^t$ and $\vec{R}_i^t$ respectively) is computed. We chose $\vec{R}_i^t$ so that the marginal utility after a successful sale is equal to that of the lowest ranked good to be bought. Similarly we chose $\vec{Q}_i^t$ so that the marginal utility after a successful purchase is equal to that of the highest ranked good being sold. This procedure assumes that $\vec{P}_i^{*t}$ and $\vec{E}_i^t[p]$ are already provided. This partition must meet the acceptance criterion:

$$\vec{R}_i^t \cdot \vec{P}_i^{*t} + \vec{y}_i^t \geq \vec{Q}_i^t \cdot \vec{E}_i^t[p] \tag{2.12}$$

If the partition is rejected a new partition is choosen and the process repeats until a partition is accepted. We are guaranteed that at least one partition must be accepted because the case where all goods are being sold trivially satisfies the acceptance criterion.

This model provides actions for four of the seven simulation stages:

Rank Goods: Compute the marginal utility, MU^t_{ij}, for each good in the economy and order the set of goods, K, by their marginal utility from lowest to highest. Let $SELL$ be the subset of K that are being sold and BUY be the subset of K that are being bought. For the current partition of goods, let $_{buy}MU^t_{ij}$ be the marginal utility of lowest ranked good being bought and $^{sell}MU^t_{ij}$ be the marginal utility of the highest ranked good being sold.

1. $SELL = \emptyset$, $BUY = K$, $_{buy}MU^t_{ij} =^{sell} MU^t_{ij}$

2.

$$\forall k_j \in SELL \quad . \quad \exists R^t_{ij} \cdot \frac{\eta_{ij}}{(E^t_i[p_j])(X^t_{ij} - R^t_{ij})} =_{buy} MU^t_{ij} \tag{2.13}$$

$$\forall k_j \in BUY \quad . \quad \exists Q^t_{ij} \cdot \frac{\eta_{ij}}{(E^t_i[p_j])(X^t_{ij} + Q^t_{ij})} =^{sell} MU^t_{ij} \tag{2.14}$$

3. If $\vec{R}^t_i \cdot \vec{P}^{*t}_i + \vec{y}^t_i < \vec{Q}^t_i \cdot \vec{E}^t_i[p]$ Then choose a new partition of K, goto step 2.

Information Diffusion: Use the slow-diffusion model of information propagation.

Update Diffusion Statistics: No action defined.

Choose Merchant: Choose the merchant with the lowest price below cutoff $(E^t_i[p_j])$

Retry Transaction: No action defined.

Update Sale Statistics: No action defined.

Calculate Expectations:

$$P^{*t+1}_{ij} = P^{*t}_{ij}(1 + \lambda_{ij}\frac{R^{*t}_{ij} - R^t_{ij}}{R^t_{ij}}) \tag{2.15}$$

$$E^{t+1}_i[p_j] = E^t_i[p_j] + \lambda_{ij}(\tilde{P}^t_{ij} - E^t_i[p_j]) \tag{2.16}$$

Chapter 3

The Cellular Automata Model

Cellular automata (CA) are well known to computer scientists and mathematicians. They have been used under many names and in many different disciplines. Toffoli and Margolus' book, *Cellular Automata Machines* [7, pages 8–11], gives an excellent summary of the history of the subject. Though CA were first introduced in the late forties by John von Neumann [9], the field remained largely dormant for nearly thirty years due to lack of communication and differing nomenclature. In 1970 an article in Scientific American by Martin Gardner about a CA model called the game of Life, invented by John Horton Conway, introduced CA to the public and a generation of young scientists. [5] The proceedings from an workshop on CA held at Los Alamos, New Mexico, provides a detailed exposition on previous uses of CA in a number of disciplines. [4]

In general, CA are dynamical systems with the following defining features:

- Discrete in space: System consists of a discrete grid of spatial cells or sites.
- Discrete states: Each cell has a finite (usually small) number of possible states.
- Discrete in time: The state of each cell is updated in a sequence of discrete time steps.
- Homogeneous: All cells are identical, and are arranged in a regular array.
- Synchronous updating: All cell states are updated in synchrony, each depending on the previous states of neighboring cells.
- Deterministic rule: Each cell state is updated according to a fixed, deterministic rule.

- Spatially local rule: The rule at each site depends only on the states of a local neighborhood of sites around it.

- Temporally local rule: The rule for the new state of a site depends only on states for a fixed number of preceeding steps (usually one).

For this project I have relaxed a few of these requirements. The agents in the economy have goods which they sell/buy and the prices/cutoff for these goods are continuous as opposed to taking one of a finite number of values. They are part of the agent's state and as such violate the discrete state requirement. This is not a major problem since the prices are stored as 32-bit floating point numbers. [1] Also when agents buy or sell goods they may do so with any other agent anywhere in the lattice. This violates the requirement of spatial locality. This was necessary to have a realistic simulation of economic transactions. In a real economy agents are not restricted to buying and selling from their immediate neighbors. We do not see this as a major problem since the stage that involves the actual transactions is the only one that violates this requirement. The bulk of BISHOP is dedicated to embodying the CA model in code that is flexible and robust. Chapter 4 gives an overview of the implementation of the CA model using data-parallel programming on the CM-2 computer.

[1] In the most general sense, the agent's state is still discrete since there is a finite (although quite large) number of values that can be represented in a computer using 32 bits.

Chapter 4

Data-Parallel Programming and the CM-2 Computer

There are two main classes of parallel computers: Single Instruction Multiple Data (SIMD) and Multiple Instruction Multiple Data (MIMD). These classes are further subdivided into shared-memory and distributed-memory architectures. Processors in a SIMD parallel computer all execute the same instruction stream but they operate on different pieces of data. Processors in a MIMD parallel computer execute distinct instruction streams on different pieces of data. Programming SIMD and MIMD machines require different methodologies. With MIMD machines, processors may be executing different instructions concurrently. If the programming problem requires intercommunication between processors some form of message passing must be devised. Often programmers wish to divide their problems into many interdependent parts. These parts can be executed on different processors. Each part depends on the results obtained from another part and the actions of the processors must be synchronized or sequenced so that delays do not occur. It is easy for deadlocks to occur when parts are co-dependent. This makes coding problems on MIMD machines difficult when there are lots of parts to deal with. With SIMD machines all the processors are synchronized to one instruction stream. Consequently the programmer knows at all times exactly what each processor is doing. The drawback is that problems cannot be divided into concurrently executing parts as in the MIMD case.

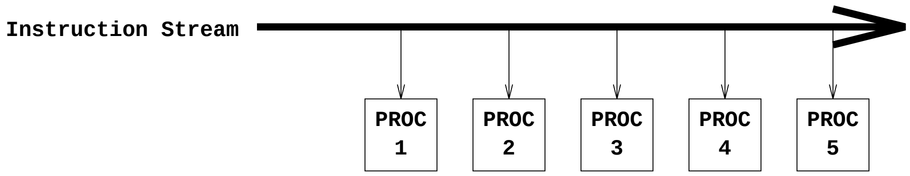

Figure 4-1: SIMD Architecture

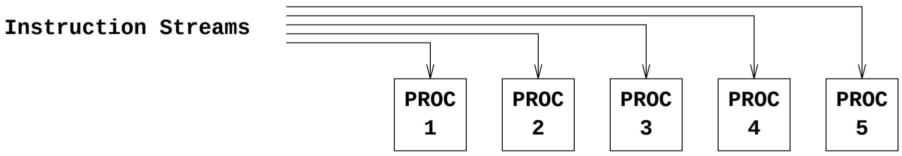

Figure 4-2: MIMD Architecture

Memory organization affects how processors communicate. Shared-memory machines have one large address space that is divided among all the processors. Some form of arbitration is necessary so that each processor can access the memory without contention and so that each processors segment memory is protected from other processors. When communication is desired processors can allocate a portion of their memory segments to be shared by other processors. Information is transfered by writing directly into another processor's memory segment. The CM-2 computer is a distributed-memory architecture. Each processor has its own memory and no other processor can access it. Communication is accomplished via an eleven-dimensional hypercube network. This eliminates the need for memory arbitration or protected memory.

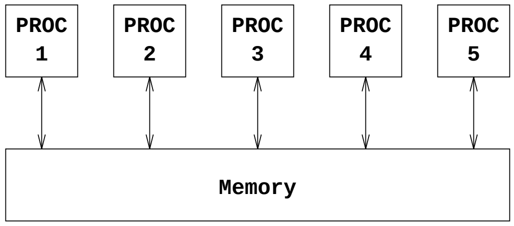

Figure 4-3: Shared Memory Architecture

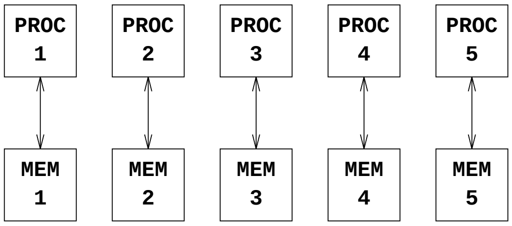

Figure 4-4: Distributed Memory Architecture

The CM-2 computer is a data-parallel, Single Instruction Multiple Data, distributed-memory architecture. The CM-2 computer is designed to work in a co-processing configuration. A UNIX front end is connected to the CM-2 computer's processors and sends instructions to them over a high speed 128-bit bus. Data can also be transfered between the front end memory and the CM-2 computer's memory which is bit-addressable. We use C*, an extension of ANSI C, to program the CM-2 computer (CM Fortran and *Lisp are also available). C* extends the data types provided by C with the `bool` type. A `bool` is an unsigned, single-bit data type. C* also extends the set of operators provided by C and also defines new behavior for the standard C operators when used with parallel variables.

4.1 The Parallel Shape

To allow the user to specify the precise way in which the data is mapped to the processors the C* programming language defines the notion of a parallel *shape*. The shape specifies the number and topology (or geometry) of virtual processors (VP). Shapes can be thought of as arrays of VPs (possibly of many dimensions). The shape used to represent the CA lattice for BISHOP has three dimensions (see Appendix B.1, bishop.hs, line 14). The total number of positions in a shape must be a power of two and a multiple of the number of physical processors in the CM. This is a current restriction of the C* programming language.

Shapes are used as part of the declaration of variables. When a variable of a particular type is declared it is allocated in parallel. There is one element of the variable for every position of the shape. The number of positions in a shape can be any integer multiple of the number of physical processors. This integer is know as the virtual processor ratio (referred to as Φ or VP ratio). Each physical processor emulates Φ VPs. For example, suppose we have a CM-2 computer with 32768 physical processors and we define a shape with 65536 positions ($\Phi = 2$). If we then declare an integer variable, call it `A`, of that shape we get 65536 elements of `A`. All the elements have the same declared name (`A`) but they each are allocated in the memory of a different VP. Each physical processor must divide its memory into Φ segments of equal length. Each has Φ elements of the integer variable in its memory. This is all organized by the compiler in a manner that is transparent to the programmer.

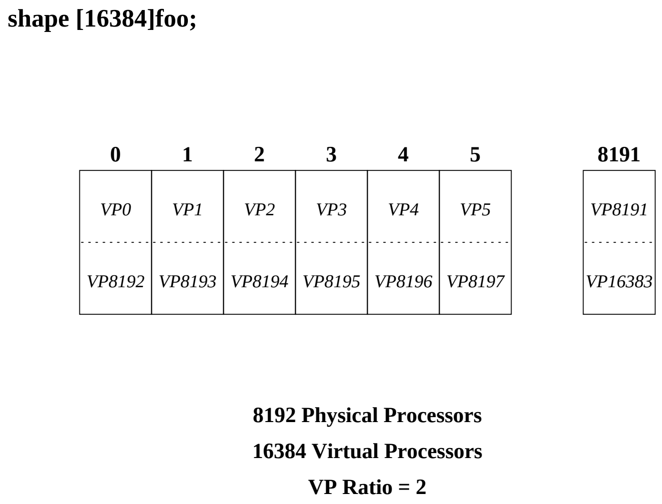

Figure 4-5: Shapes and Virtual Processors

Shapes provide an abstraction to make parallel data conform to the particular programming problem. VPs communicate based on the geometry defined by shapes. Each VP is viewed as 'residing' at a particular position in the shape. Communication between VPs is accomplished by specifying the source and destination VP's position and the data to be sent. The syntax used in C* is left-indexing. Whereas ordinary C arrays are right-indexed (`A[i]`), parallel variables are left-indexed (`[i]A`). Left-indexing can also be used to move data between the Connection Machine (CM) and the front end (FE). Programs may define many shapes, by declaring them of the type `shape`, and the `with` statement is used to select one of them to be *current*. References to parallel variables are done with respect to the current shape.

4.2 Context and the `where ... else` Statement

C* introduces the notion of *context* to simulate instruction parallelism. The SIMD nature of the CM-2 computer prevents true concurrent execution of distinct instruction streams but VPs can selectively execute instructions under program control. The `where` statement allows the programmer to turn off specific VPs based on the result of evaluating a boolean expression. In the following code fragment the expression `p1 != 0` is evaluated and the context set based on its value. VPs where `p1` doesn't equal 0 are left on while the front end sends the instruction stream for the statement `p2 = p3 / p1;`. All other VPs are turned off and do not execute the instruction stream. The `else` clause inverts the context so that VPs where `p1` equals 0 are turned on and all others are turned off while the from end send the instruction stream for the statement `p2 = 0;`.

```
where( p1 != 0 )
   p2 = p3 / p1;
else
   p2 = 0;
```

The `where` ... `else` statement may be nested, much like an `if` ... `else` statement, and the `else` clause is optional. C* also provides the `everywhere` statement which unconditionally turns on all VPs. It is most useful when nested within a `where` statement when it is necessary to temporarily suspend the effects of that `where` statement.

4.3 Parallel Assignments

C* allows three types of parallel assignments: parallel to parallel, parallel to serial and serial to parallel. In the following example every element of `A` is assigned to the element of `p2` occupying the same position. Each VP has one element of `p1` and `p2` and each VP assigns its element of `p1` to its elements of `p2`.

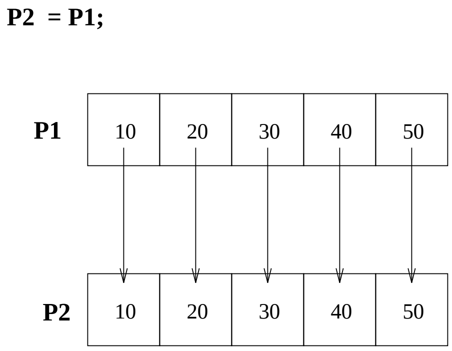

Figure 4-6: Parallel to Parallel Assignment

Serial to parallel assignments copy the same value to every element of a parallel variable. In the following example every element of `p1` is assigned the value 1.

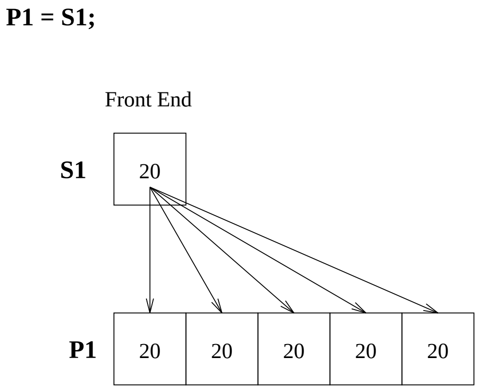

Figure 4-7: Serial to Parallel Assignment

Parallel to serial assignments reduce a parallel variable to a scalar value and assigns it to a serial variable. The value may be selected from one element of the parallel variable using left indexing, computed as the result of a reduction operation, or arbitrarily selected from one active element of the parallel variable by the C* compiler, this operation is called a *demotion*. Reduction operators combine the elements of a parallel variable into a single scalar value. The following examples demonstrate left indexing and the use of a reduction operator.

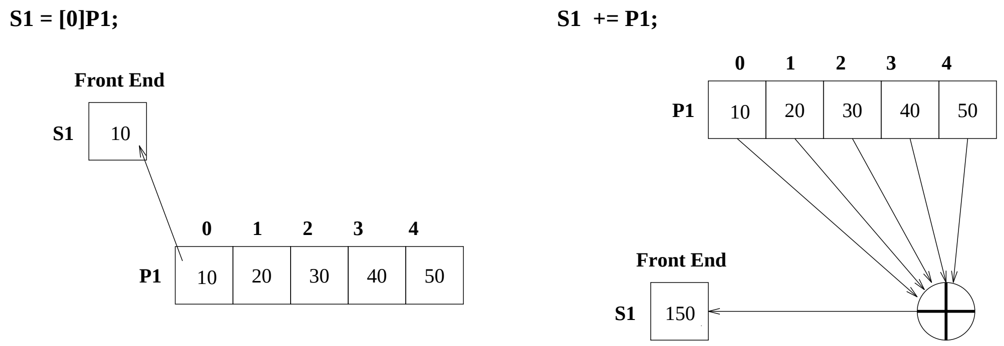

Figure 4-8: Parallel to Serial Assignments

4.4 Parallel Pointers and Arrays

C* extends the functionality of pointers and arrays into the parallel realm. Parallel pointers are themselves scalar values which reference parallel data types. Parallel arrays are in the most general sense pointers that are initialized to the address of preallocated memory. Parallel pointers and arrays can be used just as typical C pointers and arrays with the following restrictions:

- You can only do addition or subtraction with parallel pointers and arrays.
- You can add/subtract parallel variables to/from parallel pointers and arrays but only if the expression is immediately dereferenced.
- You may use right indexing with parallel pointers and arrays, using either serial or parallel variables for the index, but you cannot take the address of the resulting expression.

The following examples demonstrate how parallel pointers and arrays may be used.

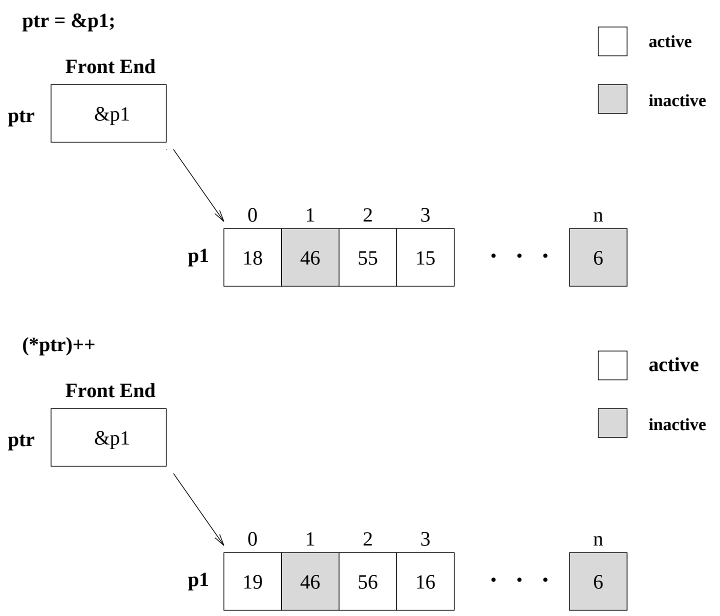

Figure 4-9: Parallel Pointers

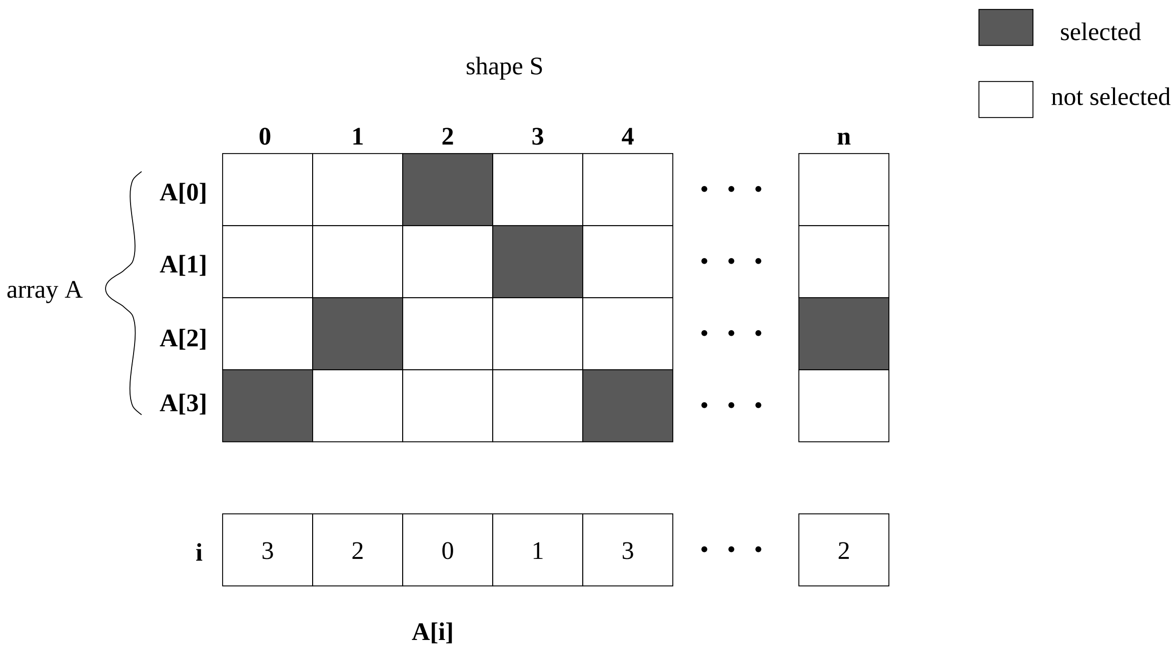

Figure 4-10: Parallel Arrays

4.5 Data Parallelism

The power of data-parallel architectures comes from the computer's ability to perform the same operation on many pieces of data. The CM-2 computer is similar to parallel vector machines that operate on vectors of data. Parallel variables are "spread" across VPs. For example, if we want to add two arrays of length 32768 and store the results in a third array on a serial computer, we could use the following code fragment:

```
int S1[32768], S2[32768], S3[32768];
int i;

for(i = 0; i < 32768; i++)
   S3[i] = S1[i] + S2[i];
```

On a very fast workstation we might say that integer addition takes 15ns. If we ignore the time it takes to update the loop variable this code segment would take $32768 * 15ns = 491.52ms$ to execute. To accomplish this on the CM-2 computer we could use the following code segment:

```
shape [32768]foo;
int:foo P1, P2, P3;

P3 = P1 + P2;
```

Using the same time measurement for integer addition this piece of code takes 15ns to execute. The `shape` statement declares `foo` with a geometry of 32768 VPs. The variables P1, P2 and P3 are declared as integers of the shape 'foo'. The 32768 elements are spread accross 32768 VPs. Each VP performs the same operation (integer addition) on a different piece of data. Each of the VPs take its copy of P1 and P3, adds them together and stores the result in its copy of P3. In this way all 32768 elements are added concurrently and no loop is needed. To exploit the power of data-parallel, SIMD architectures like that of the CM-2 computer problems must be arranged in such a way as to spread the data being operated on across VPs. Large data sets can be operated on in parallel and solutions found quickly. Of course, it is impossible to arrange all problems such that calculations on the data can be done independently for each data point. To overcome this problem the VPs in the CM-2 computer can transfer data to each other through the hypercube communication network.

4.6 Parallel Communication and the CM-2 computer

4.6.1 Grid Communication vs. General Communication, Sends and Gets

Grid and general communication are two types of data communication that can occur between VPs. Grid communication is relative to the source element, where the destination and source elements are of the same shape. General communication allows for different shapes and relative positions of the source and destination elements.

Grid Communication

Grid communication is useful when the pattern of communication doesn't change from position to position and the distance between the source and destination elements is small. It is much faster than general communication and the C* compiler can optimize grid communication. The following code fragment demonstrates grid communication where every VP 'send's a value to the VP one position higher.

```
shape [32768]foo;

int:foo source, dest;

with(foo) {
    [pcoord(0) + 1]dest = source;
}
```

The `with` statement makes `foo` the current shape. The `pcoord` function creates a parallel variable in the current shape where each element is initialized to its coordinate along the axis specified.

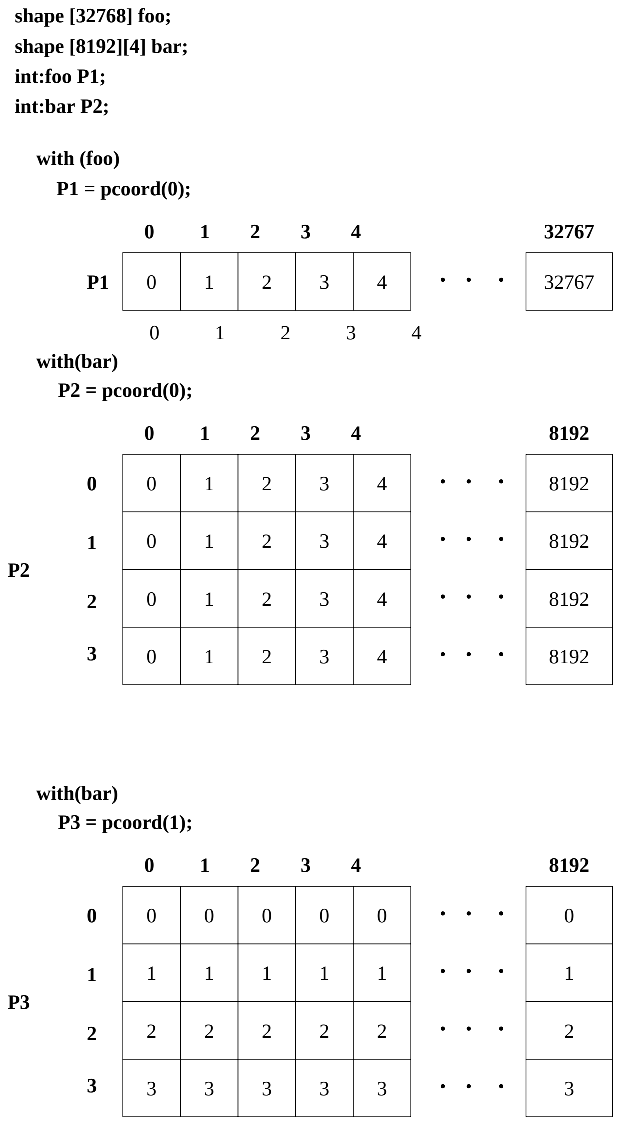

Figure 4-11: `pcoord` Function

One is added to the values returned by `pcoord` so that the source elements are mapped to the destination elements at a position one higher. The element of the source at position 32767 is mapped to the element of the destination at position 32768. This position doesn't exist (since indexing begins at 0 and ends at 32767) and the value sent is lost. Also the element at position 0 of the destination does not receive a value.

We can use the C* modulus operator (`%%`) to cause values to wrap around at the edges of the shape. The following code demonstrates this:

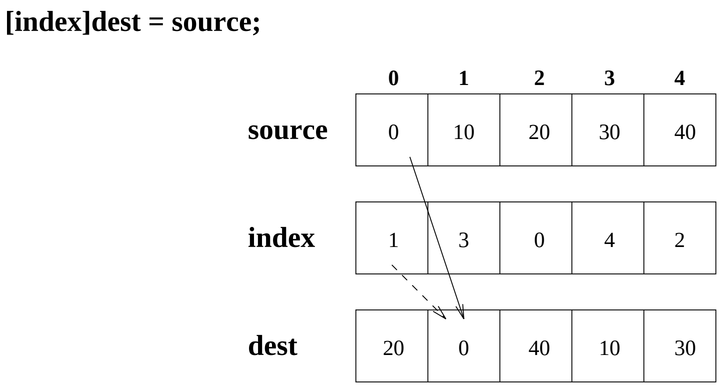

Figure 4-12: Grid Send

```
shape [32768]foo;

int:foo source, dest;

with(foo) {
    [(pcoord(0) + 1) %% 32768]dest = source;
}
```

We can accomplish the same source to destination mapping using 'get' operations instead of sends. The results are identical except at the edges of the shape.

```
shape [32768]foo;

int:foo source, dest;

with(foo) {
    dest = [pcoord(0) - 1]source;
}
```

In this case the destination elements 'get' values from source elements at positions one lower in the shape. Element 0 of the destination tries to get from element -1 of the source, which doesn't exist, and the results of the get operation are undefined for this element. Again the C* modulus operator can be used to cause values to wrap around at the edges of the shape.

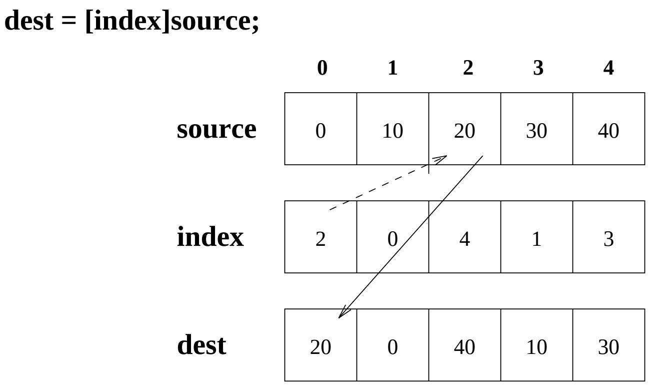

Figure 4-13: Grid Get

General Communication

General communication is used when values must be exchanged between parallel variables of differing shapes or when the pattern of communication is not static. It is accomplished using left-indexing just as grid communication is. The following code fragment demonstrates how data can be moved between variables of differing shapes:

```
shape [6]foo;
shape [3][2]bar;

int:bar dest, index;
int:foo source;

with(bar) {

    index = pcoord(0) + pcoord(1) * 3;
    dest = [index]source;
}
```

Every element of `index` is filled with a unique number between 0 and 32767. When used as a left-index for `source` this results in a bijective mapping of the elements of `source` to those of `dest`.

The mapping does not have to be bijective nor does it have to move elements in order. We could initialize `index` as follows:

```
    index = 5 - pcoord(0) - pcoord(1) * 3;
```

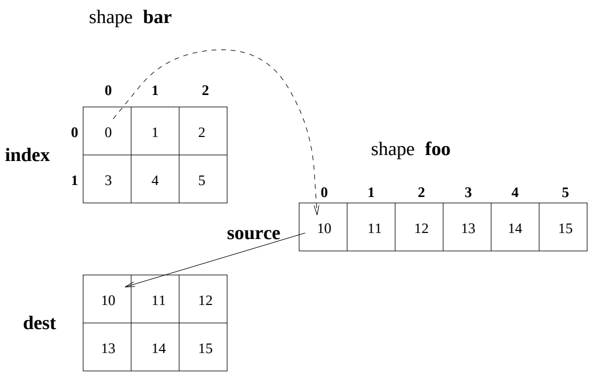

Figure 4-14: Mapping a Parallel Variable to Another Shape

This is also a bijective mapping but data is moved in reverse order as `index` contains the exact reverse of the first case.

It becomes clear that general communication can result in communication patterns that send more that one source element to a destination (**injective mapping**). This is called a collision. The default behavior is for C* to arbitrarily 'pick' one source element. Alternatively, C* allows you to specify how values are to be combined during collisions. You may take the logical or, logical and, logical exclusive or, sum, maximum, or minimum of the elements arriving at a destination.

4.6.2 Scan, Reduce, Spread, Multispread, Enumerate, Rank and Global

Complex communication patters can be created using specialized C* functions. Some C* primitives involve both communication and computation. To accomplish this C* defines the notion of a scan class. Two positions belong to the same scan class if they differ only along the specified axis. [6, pp. 174 - 177] Positions may be rendered inactive based on the current context and the active positions of a scan class are known as the scan subclass. These subclasses may be further divided into scan sets upon which operations can be performed. The functions available are the scan, reduce, spread, multispread and global functions. The functions differ in the kinds of computation that are available, the way elements are selected and the way the function returns the result of the computation.

Scan: Returns a parallel result that contains in every element of each scan set a running combination of its computations. The computation is carried out independently for each scan set.

Reduce: Returns a parallel result that contains in one element of each scan subclass the consequence of the specified computation. The computation is carried out for each scan class.

Spread: Returns a parallel result that contains in every element of each scan subclass the consequence of the specified computation. The computation is carried out for each scan class.

Multispread: For shapes of more than two dimensions C* defines the notion of hyperplanes. Two positions belong to the same hyperplane if they differ only in the specified axes. Hyperplanes are supersets of scan classes. Multispread returns a parallel result that contains in every element of each hyperplane the consequence of the specified computation. The computation is carried out for each hyperplane.

Enumerate: Returns a parallel result that contains in every element of each scan set the size of the scan set to which it belongs.

Rank: Returns a parallel result that contains in every element of each scan set a numerical ranking of the elements of the specified parallel variable. The ranking is carried out independently for each scan set and can be from lowest to highest or highest to lowest.

Global: Returns a serial result, to the front end, that is the consequences of a reduction performed on every active element of the specified parallel variable.

Chapter 5

The Command Interface

5.1 Concepts

The command interface (CI) provides the BISHOP user with a flexible command line from which he can issue COMMANDS to control the simulation, set parameters and setup statistics files for data collection. Also the CI is the interface between the CM-2 computer's *checkpointing* facilities and the DataVault, a high-speed, mass-storage disk subsystem. The user controls BISHOP by issuing commands, via the command line, that perform a number of tasks. To provide a user-friendly text interface, for the command line, we chose to use the GNU readline and history libraries [1]. The readline library provides a robust command line interface that allows the user to manipulate one line of text using the same keys that are used in GNU emacs and many popular UNIX shells. The history library is used by the readline facility to provide a history buffer of previously entered lines. Readline also provides automatic filename completion. These libraries combine to make a very smooth and flexible line interface for BISHOP. The most commonly used keys sequences that may be used for line editing are as follows:

CTRL-A Move cursor to front of line.

CTRL-E Move cursor to end of line.

CTRL-B Move cursor back one character.

[1] The readline and history libraries are available in the public domain. They can be obtained by anonymous ftp from ftp.uu.net and are subject to the GNU copyleft restrictions

CTRL-F Move cursor forward one character.

DEL Delete the character to the left of the cursor.

CTRL-D Delete the character underneath the cursor.

CTRL-_ Undo the last thing done.

CTRL-L Clear the screen, reprinting the current line at the top.

CTRL-K Kill text from the cursor to the end of the line.

CTRL-Y Yank the most previously killed text back into the line at the cursor.

TAB Search the current directory for a file(s) that begins with the word directly to the left of the cursor and complete the filename in the line.

The Connection Machine system allows the running state of programs to be saved to *checkpoint* files. The files can then be used to restart the program and continue execution as if the program had never been halted. The CI provides an interface to this facility that allows users to specify the name of the checkpoint files, schedule the program to be periodically checkpointed or schedule the program to be checkpointed once at a specific time. The CI also provides an interface to the DataVault subsystem. The bstdio library provides routines similar to the C stdio library for reading and writing files on UNIX filesystems and the DataVault. Refer to Appendix A for specifications of these procedures.

To allow the user to setup files where data from the simulation can be written we have defined the notions of VARIABLES and FILES. VARIABLES are symbolic names for data that can be written to files. Most often VARIABLES are named and refer to actual program variables. The CA skeleton provides a set of default VARIABLES that refer to the part of CA state space. FILES are symbolic names that refer to actual UNIX and/or DataVault files that can be opened for writing. FILES refer to either an UNIX file, a DataVault file or both. Users *associate* VARIABLES with a FILE and then they can request the CA skeleton to write data, from these VARIABLES to the FILES, using the bstdio library provided by the CI.

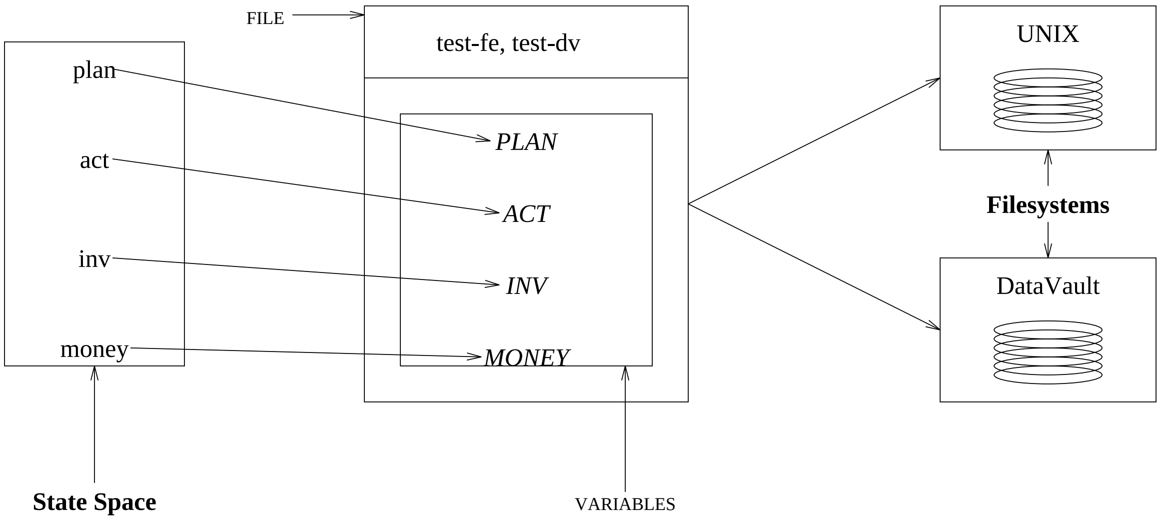

Figure 5-1: Associating VARIABLES with FILES

Users must *activate* sc variables after they have been associated with a FILE. Active VARIABLES will be written to their associated FILE. The user can deactivate a VARIABLE to temporarily prevent it from being written to the FILE.

We define the notion of an OPTION so that users may modify parameters that affect the BISHOP simulation. OPTIONS are symbolic names that refer to parameters that take values. OPTIONS may be *essential* in which case they must be given values (*set*) before BISHOP will allow a simulation to commence. Non-essential OPTIONS may be set at any time. The CA skeleton provides a set of default OPTIONS. BISHOP has two main modes of operation: command mode and simulation mode. When BISHOP is in command mode the interactive command line is active so that users can issue commands. When BISHOP is in simulation mode the command line is inactive and the automata lattice is being updated. We define the notion of a BREAKPOINT so that users may halt the simulation at key stages to examine and modify parameters. BISHOP BREAKPOINTS are similar to C debugger breakpoints. The user may activate a BREAKPOINT and when the simulation reaches it BISHOP is returned to command mode. Currently no BREAKPOINTS are available in BISHOP.

BISHOP's command line interface, coupled with the FILE, VARIABLE, OPTION and BREAKPOINT extensions, provide a simple but flexible user interface. No user interface, however, can meet all the needs of every user so the command interface is extensible. In particular users will want to add their own VARIABLES to be used with the particular model of agent behavior to be studied. Users may extend the set of COMMANDS, FILES, VARIABLES, OPTIONS and BREAKPOINTS to suit their needs. BISHOP currently can support a total of 2048 extensions. To further improve BISHOP's robustness users may define their own types of extensions [2] A library of utility procedures is provided to simplify common tasks (such as parsing command arguments). The code for BISHOP includes one very simple example of a user defined VARIABLE. It was included to provide time stamps so that rough timings could be done for each period.

5.2 Commands

This section gives a brief synopsis on the standard COMMANDS provided by BISHOP. There are 18 by default and are the minimum set of commands necessary to have a complete system. They have minimal error checking and it is assumed that the user has a good understanding of how they are used.

help With no arguments this COMMAND displays the list of available COMMANDS. Optionally it can display usage information about a particular COMMAND.

file Opens a FILE, on the UNIX front end and/or the DataVault. The user may optionally specify a flush interval which determines how many writes to the FILE must occur before any buffered data is flushed to disk.

cfile Closes an open FILE.

var Activates or deactivates a VARIABLE. When activated a VARIABLE may be written to its associated sc file. With no arguments it prints the list of VARIABLES available.

assoc Associates a VARIABLE with a FILE. A VARIABLE may be associated with at most one FILE. With no arguments is prints the list of FILES and the VARIABLES associated with them.

dassoc Disassociates a VARIABLE with a FILE. This will also deactivate a VARIABLE, if it is active, since only associated VARIABLES may be written to FILES.

[2] An extension is a general term for any functionality added to the command line. Often users want functionality that isn't a COMMAND, FILE, VARIABLE, OPTION or BREAKPOINT.

opt This COMMAND sets an OPTION, if a value is provided, or displays the current value of an OPTION. With no arguments it displays the list of available OPTIONS.

break Activates or deactivates a BREAKPOINT. With no arguments it prints the list of available BREAKPOINTS.

run Exits command mode, invokes the procedures that initialize the CA skeleton, model and utilities and begins the simulation. It initiates simulation mode.

cont Continues a simulation that has been interrupted because of a keypress or a BREAKPOINT.

source Opens the specified UNIX file and executes the COMMANDS contained in it.

check Causes an immediate checkpoint of BISHOP.

log Opens a log file, activates command logging or deactivates command logging. With no arguments it prints the status of command logging. [3]

echo Activates or deactivates command echoing from 'source'd files. With no arguments it displays the status of command echoing.

load Loads values from the DataVault into a state space variable [4].

save Saves values from a state space variable into a DataVault file [5].

reset Resets BISHOP to its startup state. It deallocates all memory used on the CM, closes all FILES, disassociates all VARIABLES and rereads the startup file. A "run" COMMAND encountered in the startup file during a reset is ignored.

quit Frees all the memory allocated on the CM, closes all FILES and exits BISHOP returning the user to the UNIX shell.

[3] This command currently does nothing since the command logging facility has not yet been implemented.
[4] This facility is not yet available
[5] This facility is not yet available

Chapter 6

Coding the Cellular Automata Skeleton

6.1 Data Structures and Global Variables

The heart of BISHOP is the CA skeleton. It is the link between the model of agent behavior, the command interface and the CA framework. BISHOP's flexibility comes from a general notion of an economy and the actions that go on during a period of economic activity. The seven stage process outlined in chapter 1 totally defines one period of activity. The CA skeleton is responsibly for guiding the simulation through those seven stages each period and maintaining the CA state space. A number of global variables and procedures are used for these purposes.

```
extern shape [][][]lattice;

/* Transaction bookkeeping */
typedef struct trans {
  bool chng;
  unsigned char disp;
  unsigned char flag;
  int who;
  int quant;
  float price;
} TRANS;

extern float:lattice *money;
extern int:lattice *inv;
```

```
extern TRANS:lattice *plan;
extern TRANS:lattice *act;
extern bool:lattice *hit;

extern int LS;
extern int G;
extern int maxiter;

extern int period;

void init_bishop();
void init_model();
void init_util();
void finish_bishop();
void finish_model();
void finish_util();

void try_transaction();
void choose_merchant();

/* Model hooks */
void (*diffuse_info)();
void (*rank_goods)(), (*update_diff_stats)();
void (*update_sale_stats)();
void (*calculate_expectations)();
void (*retry_transaction)();
```

The global variables `G` and `LS` hold the number of goods in the economy ($\mathcal{G}$) and the lattice size ($\mathcal{LS}$) respectively. The global variable `maxiter` holds the number of periods for which the simulation runs. The CA state space is held in the global variables `plan`, `act`, `money` and `inv`. Each agent is required to fill in the details of their plan for the current period in `plan`. Each agents inventory of goods and supply of money is held in `inv` and `money` respectively. These variables are updated by the CA skeleton in the `try_transaction` procedure. Each stage of the simulation has a related procedure or procedure hook. The "choose merchant" stage is not optional so it has a procedure. All other stages are optional and so they have a procedure hook. When the user codes a model of agent behavior these hooks must be set to the addresses of the proper procedures. All hooks that have been set will be called by the CA skeleton during the appropriate stage.

6.2 The main Procedure

The `main` procedure for the program is very simple. The various parts of the program are initialized to their preliminary states. Initialization occurs in two stages. The preliminary stages sets the default values of global variables and flags. The final stage completes all tasks necessary for a simulation to begin. This includes allocating dynamic data structures on the CM-2 computer and the front end, opening files, etc.

```
main(int argc, char *argv[])
{
  OPTSPEC os;

  printf("BISHOP v5.0 Cellular Automata Simulation\n");
  printf("by: Harris L. Gilliam (May 18, 1993)\n");
  model_intro();
  printf("\n\n\n");

  /* Init command interface */
  init_command(argc, argv);

  /* Do prelim init of bishop skeleton */
  init_bishop(argc, argv, PRELIM);

  /* Do prelim init of model */
  init_model(argc, argv, PRELIM);

  /* Do prelim init of utilities */
  init_util(argc, argv, PRELIM);

  /* Read commands in startup file */
  startup(0);

  /* This setjmp will be used by a longjmp in c_reset */
  setjmp(bishop_env);

  /* Parse commands until we have a lattice size */
  do {
    parse_cmndline();
    do_option(ELS, GETOPT, &os);
  } while(!os.set);

  with(lattice) {

    /* Parse commands until user quits */
```

```
    while(1) parse_cmndline();
  }
}
```

The procedure `startup` reads the startup file and any commands contained in it are executed. A `setjmp` call is issued to mark the place where program execute will return to when a **reset** command is issued. The `do` loop uses the command interface (CI) to parse and execute a command line from the user. The call to `do_option` retrieves the current value of `LS` and if this OPTION has not been set then we continue to parse commands from the user. Once it has been determined that `LS` has been given a valid value we know that the shape `lattice` has been allocated and can be used. All that remains is to make `lattice` the current shape and parse commands from the user until he/she issues a **quit** command.

6.3 The `try_transaction` Procedure

When a user issues a **run** command the CA skeleton invokes the `run_bishop` procedure. This procedure controls the flow of the simulation. No complex algorithmic issues arose during its coding and the details of how it can be clearly seen by reading the source code. The most interesting aspect of the CA skeleton is the `try_transaction` procedure. This procedure is invoked after the "choose merchant" stage at which point every agent has devised a plan for each good in the economy. The CA skeleton attempts to put these plans into action. Typically there will be many consumers for each merchant in the economy. The CA skeleton arbitrates between consumers that want access to the same merchant. Consumers are given preference by geographic proximity. Let X_m, Y_m and Z_m be the x, y, and z coordinates of the merchant in the lattice. Let X_c, Y_c and Z_c be the x, y, and z coordinates of the consumer in the lattice.

$$D = \max\left[\left|X_m - X_c\right|, \left|Y_m - Y_c\right|, \left|Z_m - Z_c\right|\right] \tag{6.1}$$

Each merchant has a list of potential consumers. The consumers are allowed to transact with the merchant based on their proximity with closer consumers transacting first. If two or more consumers are the same distance from a merchant then one consumer is randomly selected. Upon completion of the transaction the consumer is removed from the list of potential consumers for that merchant and the arbitration process repeats. This continues until all consumers have had access to the merchant's they desire to do business with.

```
typedef struct key_S {
  int c_addr;
  int rndc;
  int dist;
} KEY;


void try_transaction(int j)
{
  int:lattice xi, yi, zi;
  int:lattice xd, yd, zd;
  bool:lattice done = 0;
  KEY:lattice key;
  KEY:lattice pick;
  int:lattice sa;


  where(plan[j].who != NO_ONE)
    if (CM_global_count_context() != 0) {

    g_coord(plan[j].who, &xi, &yi, &zi, LS);
    xd = abs(XS - xi);
    yd = abs(YS - yi);
    zd = abs(ZS - zi);

    key.dist = (xd > yd ? (xd > zd ? xd : zd) : (yd > zd ? yd : zd));
    key.rndc = prand();
    key.c_addr = SELF_ID;
```

The heart of the algorithm used in `try_transaction` involves the `KEY` structure. When the `try_transaction` procedure is called the context has been restricted to those agents who are buying the good indexed by `j`, ie. consumers. The merchant's id, from whom the consumer wants to buy, is held in `plan[j].who`. The variable `key` is filled with the distance to the merchant, a random number, and the customer's id.

```
    /* Make send address from news coordinates */
    CM_make_news_coordinate_1L(CM_vp_set_geometry(lattice),
       &sa, 0, &xi, 32);
    CM_deposit_news_coordinate_1L(CM_vp_set_geometry(lattice),
 &sa, 1, &yi, 32);
    CM_deposit_news_coordinate_1L(CM_vp_set_geometry(lattice),
```

```
 &sa, 2, &zi, 32);

    while ((&= done) == 0) {

      /* Init pick to a value larger that any customers distance */
      everywhere
pick.dist = LS + 1;

      where (done == 0) {
/* Send with min */
CM_send_with_u_min_1L(&pick, &sa, &key, boolsizeof(key), CM_no_field);

where([xi][yi][zi]pick.c_addr == SELF_ID) {
```

The arbitration between consumers relies on general communication with computation. The `key` structure for each consumer is sent, using the `CM_send_with_u_min_1L` procedure, to the desired merchant, with the variable `pick` as the destination, as 96-bit unsigned integer number. A minimum combiner is applied to them to select the smallest value. The fields of the structure are interpreted by the combiner as one integer with the `dist` member occupying the most significant bits. The `rndc` member occupies the next most significant bits and the `c_addr` member occupies the least significant bits. The combiner will select the smallest 96-bit integer and the `dist` member will have the greatest influence. This operation will pick out the consumer that is the smallest distance away from the merchant. The random number held in `key.rndc` serves to differentiate consumers that are the same distance from the merchant.

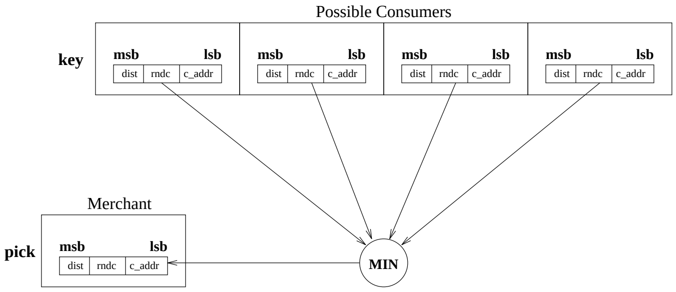

Figure 6-1: Consumer Arbitration

The value stored in the `pick` variable for the merchant is compared to the consumer's id. The context is restricted to those consumers whose id's match and the transaction is initiated. The variable `done` is used as flag to mark consumers that have been selected by the arbitration process so that they will be excluded in future rounds.

6.4 Options and Variables

The CA skeleton provides a number of OPTIONS and VARIABLES for the user. All of these OPTIONS are essential and must be set before a simulation can begin. The VARIABLES allow the portion of state space maintained by the CA skeleton to be written to statistics files.

Options

LS The size of the CA lattice.

G The number of goods in the economy.

maxiter The number of periods the simulation is to run for.

Variables

PBEG A beginning of period marker.

PEND An end of period marker.

PLAN The plans devised by each agent (held in the `plan` global variable).

ACT What actually happened with each agent (held in the `act` global variable).

WLTH Each agents inventory of goods and cash reserves (held in the `wealth` global variable).

INV Each agents inventory of goods (held in the `inv` member of the `wealth` global variable).

MONEY Each agents cash reserves (held in the `money` member of the `wealth` global variable).

These OPTIONS and VARIABLES are registered with the CI in the `init_bishop` procedure. The global variables `defopts` and `defvars` hold the codes returned by the CI for these extensions. Macros are provided so that they may be referred to by symbolic names (see Appendix B.1, file bishop.hs, lines 41-54).

Chapter 7

Coding Economic Models

Economic models can be very complex, like those involving interactions between many economic institutions, or extremely simple, like the pure exchange economy described in chapter 2. Coding these models almost always presents a significant challenge. Often there are many parameters to be considered and the equations involving these parameters are fairly complex. Economic models always include a decision making component. The problem of making "good" economic decisions is inherently difficult because economic decisions are usually made under conditions of great uncertainty. Much of computer science is dedicated to finding optimal solutions and algorithms for decision problems. Writing code in a data-parallel language adds yet another level of complexity to an already difficult problem. None the less interesting economic models can be coded because the problem can be narrowed enough that near-optimal solutions can be found. Programming in parallel requires a change in mindset but often an algorithm can be found that finds the solution significantly faster. A number of simplifications were chosen to make the task of coding the Hedonistic Dain Bramaged (HDB) and Marginal Utility Ranking (MU) models easier. The most significant are the Free Exchange Economy and Nearest Neighbor Diffusion.

7.1 The Free Exchange Economy

The Free Exchange economy was chosen as the environment for the HDB and MU models because it simplifies the decision process. The basic decisions that agents make each period are:

- For each good in the economy: do I buy or sell this good ?

- For each good I am buying what is the most I am willing to pay (cutoff) and who do I buy from ?
- For each good I am selling what is the price I will ask and who do I sell to ?
- After I have tried to buy or sell a good how do I adjust my beliefs about my economy so I can make better choices next time ?

In a pure exchange economy goods are durable and as such the choice of which goods to sell and which to buy does not have to take production or consumption into consideration. The production and/or consumption of goods would complicate the decision because agents would have to consider rates of production, production costs and rates of consumption. In a pure exchange economy goods are conserved and agents are guaranteed that a good will not become a scarce commodity because of underproduction or overconsumption. Pure exchange economies may also contain a money commodity. The role of money however is greatly reduced. Typically an agents wealth would be measured in terms of inventory of valuable goods and cash reserves. In a pure exchange economy goods are valued with respect to the money commodity. Agents do not seek to bolster their cash reserves but balance their planned purchases with their planned sales. Whatever cash the possess gives them the ability to sell fewer goods to cover their purchases. Wealth is calculated based on the market value of each good and the agents inventory. The implementation of the pure exchange economy requires no code and actually reduces the amount of code that must be written. No code need be written to handle production or consumption of goods or money.

7.2 Information Propagation

The other simplification, the nearest neighbor diffusion algorithm, is an adaptation of an algorithm for simulating diffusion on the CAM6 [7, pp. 155-159]. Once an agent has set prices for goods to be sold this information diffuses into the surrounding neighborhood. The task was to find an algorithm for diffusion that was easy to code in C* and used local interactions between agents. Algorithms that use local interactions execute faster on the CM-2 computer because the compiler can optimize nearest neighbor communication.

Each agent maps to a VP that is part of the three axis C* shape, called `lattice` in the code, used to represent the economy. Agents may be identified by their exact coordinates in the shape, by relative offsets from one specific agent or by a unique 32-bit id computed from their coordinates in the shape. Each agent is the origin of a three-axis, cartesian-coordinate system. Agents have 27 neighbors each which are identified by their relative cartesian coordinates, called offsets.

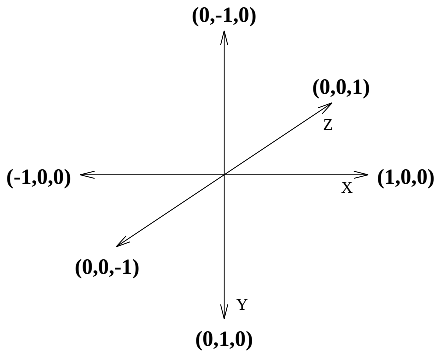

Figure 7-1: The Neighborhood

7.2.1 Cribsheets

Each agent keeps information that has diffused to him from merchants (other agents who are selling a good) on a cribsheet. The size of the cribsheet is a parameter controlled by the user ($\mathcal{CS}$). Each entry in the cribsheet contains an integer id that uniquely identifies the merchant and a set of boolean flags (the size of this set is determined by $\mathcal{G}$). When a flag is set it indicates that the agent has access to price information for that good from the agent identified by the id. There are three global variables that make up the cribsheet for each VP. `*crib_count` is the number of filled entries in the crib sheet. When `*crib_count` is equal to $\mathcal{G}$ (represented in the code by the global variable `G`) the crib sheet is full. `*crib_id` is the unique integer id for each merchant represented in the cribsheet. Finally `crib_g` is an array of booleans that serve as the flags.

```
extern int:lattice *crib_count;
extern int:lattice *crib_id;
extern bool:lattice *crib_g;
```

Everytime an agent gets price information from a merchant he searches his cribsheet for an entry for that merchant. If one is found he sets a boolean flag for the particular good. If no entry is found one is added to the cribsheet for this merchant. When the cribsheet fills up the agent must discard the newly obtained information.

7.2.2 Random Diffusion

Each agent picks a random number between 1 and ρ (where ρ is a parameter chosen by the user representing the diffusion rate, `drate` in the code) for each of his immediate neighbors. The agent updates his cribsheet with information from each neighbor for whom he picked the number one. This part of the diffusion algorithm is handled by the procedure `slow_diffusion`.

```
void slow_diffusion()
{
  register int xo, yo, zo;
  register int j;

  *crib_count_cap = *crib_count;

  for (j = 0; j < G; j++) {

    for (xo = -1; xo < 2; xo++)
      for (yo = -1; yo < 2; yo++)
        for (zo = -1; zo < 2; zo++) {
```

The variables `xo`, `yo` and `zo` are the offsets that identify neighbors. The variable `j` indexes goods. The variable `crib_count_cap` is assigned a copy of the current value of `crib_count`. It will be used in other procedures to keep track of what information was already on the cribsheet and what information is newly added. This will prevent agents from copying the new information on their neighbors cribsheet.

```
          if ((xo == 0) && (yo == 0) && (zo == 0))
            continue;

          *hit = (srnd(drate) == 1);

          where (*hit == 1) {
            get_his(xo, yo, zo, j);
```

The $(0, 0, 0)$ offset is skipped as it refers to the agent. The procedure `srnd` takes a short as an argument and returns a random number between 1 and the value passed to it. The global variable `hit` is a parallel pointer to bool. It is used in many procedures throughout the code to hold the result of boolean expressions. In this case we compare the value returned by `srnd` to 1. `*hit` will equal 1 (true) in every VP where `srnd` did return 1 and 0 (false) otherwise. The `where` statement turns off all VPs where `*hit` is false. The procedure `get_his` is called and will copy the price information, from the neighbor whose offset is stored in `xo`, `yo` and `zo`, for the good indexed by `j`.

```
#define NMAP(v, xo, yo, zo) [(.+xo)%%LS][(.+yo)%%LS][(.+zo)%%LS]v

            where(NMAP((*crib_count), xo, yo, zo) > 0)
              get_sheet(xo, yo, zo, j);
          }
        }
  }
}
```

We need to copy the cribsheet entry from the neighbor only if he is selling good `j`. The macro `NMAP` maps the relative offsets into x, y and z coordinates in the shape. It uses the `%%` operator to accomplish wraparound at the edges of the shape. We turn off all VPs where the **selected** neighbor's cribsheet is empty. The procedure `get_sheet` is called and will copy the neighbor's cribsheet entry for good `j`.

The procedures `get_his` and `get_sheet` are used by `slow_diffusion` to update the agent's cribsheet with the neighbor's price information and the neighbor's cribsheet respectively.

```
void get_his(int xo, int yo, int zo, int g)
{
  register int j, k;
  bool:lattice f = 0;


  /* If this guy has a full crib_sheet then oh well he looses out */
  /* Also if the neighbor is not selling then there is nothing to get */
  where((*crib_count != CS) && (NMAP(plan[j].price), xo, yo, zo == SELL)) {
    /* Search for duplicates */
    for (k=0; k < CS; k++) {

      where ((f == 0) & (*crib_count > k))
        where (crib_id[k] == MAPID(xo, yo, zo)) {
```

```
          crib_g[k*CS+g] = 1;
          f = 1;
        }
```

The get_his procedure first turns off VPs whose neighbors are not selling good j and VPs whose cribsheets are full. The boolean flag f is set when a cribsheet has been updated. We further restrict the context to VPs where f is 0 and *crib_count $>$ k. This will turn off all VPs whose cribsheets have been updated or whose cribsheets have less than k entries. We restrict the context to VPs where the current cribsheet entry belongs to the neighbor identified by the offsets xo, yo and zo. The MAPID macro computes a the neighbor's unique id from the offsets. The remaining active VPs set the flag corresponding to the good indexed by g and update f.

```
      /* No duplicates so we must add a new entry to the cribsheet */
      where ((f == 0) & (*crib_count == k)) {
for(j = 0; j < G; j++)
  crib_g[k*CS+j] = 0;

crib_id[k] = MAPID(xo, yo, zo);
crib_g[k*CS+g] = 1;
(*crib_count)++;
f = 1;
      }

      if (&= f)
break;
    }
}
```

This section of code handles the case where we must add a completely new entry to the cribsheet. When *crib_count equals k we know that the end of the cribsheet has been reached and no duplicate entry for this neighbor has been found. Also the kth entry is the next available one so we initialize all the flags to zero, fill the entry with the neighbor's unique id and set the flag for the good indexed by g. We use the parallel, logical and, reduction operator to reduce the parallel flag f to a single true/false value. When all of the cribsheets are updated &= f will be "true" and we break out of the loop.

7.3 Hedonistic Dain Bramaged

The code for the hedonistic dain bramaged model is for the most part a direct translation of the defining equations of the model as laid out in section 2.6. No effort was made to optimize the code and the simplest solutions were used. Of some interest is the code for the "choose merchant" stage of the simulation.

```
void choose_merchant()
{
  register int j;
  int who;
  float lowp;

  /* choose agent with the lowest price <= my cutoff */
  for (j = 0; j < G; j++) {

    plan[j].flag = HIGH_PRICES;
    plan[j].who = NO_ONE;
```

The first task is to set the plan flag to indicate failure to choose a merchant due to high prices. This flag will be overwritten for every VP where a merchant is found. Also, we initialy buy from no one.

```
    where((plan[j].disp == SELL) & (plan[j].price == seen[j].low))
      who = (int) SELF_ID;
```

Each agent has recorded the lowest price available for every good and we want to find the agent(s) who is(are) offering the good at that price. We restrict the context to VPs where the plan is to sell the good, this selects all the merchants, and to where the price of the good is the lowest price. The `SELF_ID` macro computes the agents unique id for each VP. We use a parallel to serial assignment to select one of the values from the remaining active VPs. In this way we get the id for one agent who is selling at the lowest possible price. Ideally we would want to select the agent nearest us but, since this model assumes stupidity, any agent will do.

```
    where((plan[j].disp == BUY) & (plan[j].price >= seen[j].low)) {
      plan[j].flag = NADA;
```

```
      plan[j].who = who;
      plan[j].price = seen[j].low;
      plan[j].quant = desire[j];
    }

  } /* for (j = 0; */
}
```

Finally we fill in the rest of the plan with who to buy from, at what price and how much to buy.

7.4 Marginal Utility Ranking

The code for the marginal utility ranking model is simple from a computational point of view. The effort comes in finding a simple algorithmic solution. The most complex problem involved is finding a partition of goods that meets the acceptance criterion (equation 2.12). The algorithm used is to define an parallel integer array, of length $\mathcal{G}$, that will be used to hold the indices to goods after they have been ranked by their marginal utility. Typically sorting an array is a simple task. UNIX machines are provided with the `qsort` procedure to quickly sort arrays of arbitrary length with elements of arbitrary size. Our solution is to use a procedure similar to `qsort` to sort the array of indices.

```
void rank_mu()
{
  register int j, k;
  TRANS:lattice *oplan;
  int:lattice *r;
  bool:lattice found = 0;
  int:lattice selling = 0;
  float:lattice ROP, QEP;


  /* selling marks the boundry between goods being sold and bought */
  /* goods ranked from position 0 to selling - 1 are being sold */
  /* goods ranked from position selling to G - 1 and being bought */

  /* *r is an array of integers that represent the ranking of goods by MU */
  /* r[i] holds the index for the good ranked at the ith position */

  r = (int:lattice *) PMalloc(G * boolsizeof(int:lattice));
  oplan = (TRANS:lattice *) PMalloc(G * boolsizeof(TRANS:lattice));
```

```
  boolcpy((bool:lattice *) oplan, (bool:lattice *) plan,
  G * boolsizeof(TRANS:lattice));

  /* Calculate Marginal Utility per unit expected price for all goods */
  for (j = 0; j < G; j++) {
    MU[j] = eta[j] / (EP[j] * inv[j]);
    r[j] = j;
  }

  /* Write MU */
  do_variable(EMU, STATVAR);

  /* Sort from lowest to highest */
  pbsort((bool:lattice *) r, G, boolsizeof(int:lattice), comp_mu);
```

We compute the marginal utilities for each good and assign the indices for those goods to r. Initially the array is unsorted. The procedure pbsort does a parallel bubblesort on the array. It uses the procedure comp_mu to order the elements of the array. Initially we coded a parallel version of the quicksort algorithm. We chose to go with bubblesort, however, because for short arrays bubblesort performs almost as quickly as quicksort. Also bubblesort was simpler to code. The procedure comp_mu compares two integer elements from the r array. The pbsort routine passes parallel pointers to these elements and expects a parallel integer return value.

```
int:lattice comp_mu(int:lattice *r1, int:lattice *r2)
{
  int:lattice a = 0;

  where(MU[*r1] < MU[*r2])
    a = -1;

  where(MU[*r1] > MU[*r2])
    a = 1;

  return(a);
}
```

The procedure comp_mu uses the values of the elements as indices into the MU array. For each VP the return value will be -1, 0 or 1 depending on whether the marginal utility for the good indexed by *r1 is less than, equal to or greater than the marginal utility for the good indexed by `*r2`. Sorting `r` in this way gives us a ranking of all goods by marginal utility, as required. To find a valid partition we start by buying all goods and selling none.

```
  /* Try partitions until we find one that works for every agent */
  /* The first one being to sell no good and buy all goods */
  do {

    /* Find Q and R vectors */
    where (!found) {
      construct_QR(r, &selling);

      /* Is this a feasible plan ? */
      ROP = QEP = 0.0F;
      for (k = 0; k < G; k++) {
where(plan[k].disp == SELL)
  ROP += plan[k].quant * OP[k];
where(plan[k].disp == BUY)
  QEP += plan[k].quant * EP[k];
      }

      /* Yes so we have found a partition that works */
      where((ROP + *(money)) >= QEP)
found = 1;

      /* Otherwise try selling one more good and buying one less */
      else
selling++;

    } /* where (!found) */

    ckpt_periodic(feprefix, cmprefix);

  } while ((&= found) == 0);

  for( k = 0; k< G; k++) {
    where((oplan[k].disp == SELL) & (plan[k].disp != SELL))
      plan[k].chng = 1;

    where(plan[k].disp == SELL)
      plan[k].price = OP[k];

    where(plan[k].disp == BUY)
      plan[k].price = EP[k];
  }

  PFree(r);
```

```
  PFree(oplan);
}
```

The variable `selling` holds the index into the `r` array and identifies the boundary between goods being sold and goods being bought.

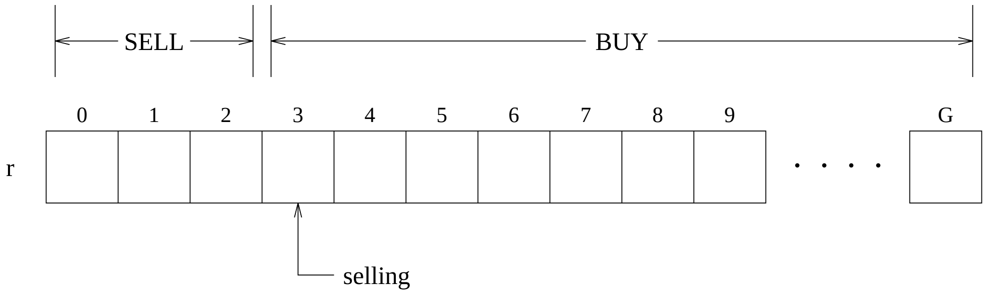

Figure 7-2: Partition of Goods Marked by `selling`

The procedure `construct_QR` is called to find values for $\vec{Q}_i^t$ and $\vec{R}_i^t$ that satisfy equations 2.14 and 2.13. Values are computed for `ROP` ($\vec{R}_i^t \cdot \vec{P}_i^{*t}$) and `QEP` ($\vec{Q}_i^t \cdot \vec{E}_i^t[p]$) from the vectors chosen by `construct_QR`. We restrict the context to those VPs where the partition passes the acceptance criterion and mark them by setting the boolean flag `found` to 1. For all other VPs we increment `selling` by 1, which creates a new partition. It moves the lowest ranked good, from the set of those being sold, to the set of goods being bought. We continue this process until a valid partition is found for all VPs. The final step is to fill in the prices and cutoffs.

Chapter 8

Using BISHOP

8.1 Setting Up a New Model

Users must compile a version of BISHOP for each model of agent behavior to be studied. In future releases of the software this will no longer required. The code for the agent model must be linked with the command interface (CI) and cellular automata (CA) skeleton to get a working executable.

8.2 Running the Program

Users execute BISHOP from a UNIX shell and begin a *session* that typically involves setting parameters, opening FILES and associating VARIABLES with them and running a simulation. When BISHOP first starts up the users see the program banner which specifies the version number, the model of agent behavior currently linked being used and the values of the available OPTIONS. BISHOP has two primary modes of operation: command mode and simulation mode. BISHOP comes up in command mode where the user can use the interactive command line.

```
BISHOP v5.0 Cellular Automata Simulation
by: Harris L. Gilliam (May 18, 1993)
MODEL:  Hedonistic Dain Bramaged


Options available:
```

```
LS: unset (lattice size) *
G: unset (# goods in economy) *
MAXITER: unset (# of periods to run simulation) *
FEP: /home/cm/guest/hackeron/bishop/checkpoints/ (front end checkpoint prefix) *
CMP: soggy-fibers:/hackeron/checkpoints/ (CM checkpoint prefix) *
CP: unset (periodic checkpoint time)
CA: unset (alarm checkpoint time)
CS: unset (cribsheet size) *
DRATE: unset (diffusion rate) *

(0:63992) CMND>>
```

The command prompt shows the current period of simulation and the amount of memory, expressed in bits, available on the CM-2 computer.

8.2.1 Setting Up Parameters

The first thing the user must do is set the OPTIONS listed marked by an asterix. These are essential OPTIONS and BISHOP cannot enter simulation mode without valid values for them. Some of theOPTIONS in this example are needed by the CA skeleton and others by the agent model. The CA skeleton needs:

LS This parameter controls the size of the automata lattice ($\mathcal{LS}$). There will be a total of $\mathcal{LS}^3$ cells in the lattice arranged in a $\mathcal{LS}$-cube.

G This parameter controls the number of goods in the economy.

FEP This parameter specifies the directory on the front end that will be used when checkpointing BISHOP.

CMP This parameter specifies the directory on the DataVault that will be used when checkpointing BISHOP.

MAXITER This parameter controls the number of periods for which the simulation will run.

The OPTIONS needed by the agent model in this example are:

CS This parameter controls the size of the cribsheets used, by the slow diffusion algorithm, for information propagation.

drate This parameter controls the rate of diffusion (ρ). Each neighbor of a merchant will have a $\frac{1}{\rho}$ chance in getting price information from the merchant.

Once all of the essential OPTIONS have been set the user can begin a simulation. To simplify the task of setting OPTIONS the user may place COMMANDS in a startup file, named "bishop.startup", that should be located in the same directory that BISHOP is started up from. Here is a startup file for this example:

```
opt LS 16
opt G 20
opt MAXITER 10
opt CS 32
opt DRATE 512
```

This file contains all the COMMANDS that the user would normally type at the command line to set the essential OPTIONS. Any COMMAND that is available at the command prompt may be used in the startup file. Here is the banner for BISHOP with the above startup file:

```
BISHOP v5.0 Cellular Automata Simulation
by: Harris L. Gilliam (May 18, 1993)
MODEL:  Hedonistic Dain Bramaged




Reading commands in startup file: bishop.startup

opt LS 16

opt G 20

opt MAXITER 10

opt CS 32

opt DRATE 512

Options available:

LS: 16 (lattice size) *
G: 20 (# goods in economy) *
MAXITER: 10 (# of periods to run simulation) *
```

```
FEP: /home/cm/guest/hackeron/bishop/checkpoints/ (front end checkpoint prefix) *
CMP: soggy-fibers:/hackeron/checkpoints/ (CM checkpoint prefix) *
CP: unset (periodic checkpoint time)
CA: unset (alarm checkpoint time)
CS: 32 (cribsheet size) *
DRATE: 512 (diffusion rate) *

(0:63988) CMND>>
```

8.2.2 Setting Up Data Collection

The **file** COMMAND is used to open FILES for data collection. VARIABLES must then be associated with the FILES, using the **assoc** COMMAND, and activated so they will be written to the FILE, using the **var** COMMAND. Below is an example of how FILES and VARIABLES are used in BISHOP. The COMMANDS were executed from a startup file.

```
BISHOP v5.0 Cellular Automata Simulation
by: Harris L. Gilliam (May 18, 1993)
MODEL:        Hedonistic Dain Bramaged

Reading commands in startup file: bishop.startup

opt LS 32

opt G 20

opt MAXITER 10

opt CS 32

opt DRATE 512

file hdbtest 1_hdbtest_10 1_hdbtest_10 10

file: opened file (soggy-fibers:/hackeron/statistics/1_hdbtest_10,
/home/cm/guest/hackeron/bishop/statistics/1_hdbtest_10) as hdbtest
with flush interval of 10

var

Statistics variables available:
PBEG
PLAN
ACT
```

```
WLTH
INV
MONEY
PEND
LS
G
MAXITER
FEP
CMP
CP
CA
CS
DRATE
DESIRE
TIME

assoc LS hdbtest

assoc: variable LS associated with file hdbtest

assoc MAXITER hdbtest

assoc: variable MAXITER associated with file hdbtest

assoc G hdbtest

assoc: variable G associated with file hdbtest

assoc PBEG hdbtest

assoc: variable PBEG associated with file hdbtest

assoc WLTH hdbtest

assoc: variable WLTH associated with file hdbtest

assoc PLAN hdbtest

assoc: variable PLAN associated with file hdbtest

assoc ACT hdbtest

assoc: variable ACT associated with file hdbtest

assoc PEND hdbtest

assoc: variable PEND associated with file hdbtest
```

```
assoc CS hdbtest

assoc: variable CS associated with file hdbtest

assoc DRATE hdbtest

assoc: variable DRATE associated with file hdbtest

assoc DESIRE hdbtest

assoc: variable DESIRE associated with file hdbtest

var LS on

var: variable LS activated

var MAXITER on

var: variable MAXITER activated

var G on

var: variable G activated

var PBEG on

var: variable PBEG activated

var WLTH on

var: variable WLTH activated

var PLAN on

var: variable PLAN activated

var ACT on

var: variable ACT activated

var PEND on

var: variable PEND activated

var CS on

var: variable CS activated
```

```
var DRATE on

var: variable DRATE activated

var DESIRE on

var: variable DESIRE activated

Options available:

LS: 32 (lattice size) *
G: 20 (# goods in economy) *
MAXITER: 10 (# of periods to run simulation) *
FEP: /home/cm/guest/hackeron/bishop/checkpoints/ (front end checkpoint prefix)
CMP: soggy-fibers:/hackeron/checkpoints/ (CM checkpoint prefix) *
CP: unset (periodic checkpoint time)
CA: unset (alarm checkpoint time)
CS: 32 (cribsheet size) *
DRATE: 512 (diffusion rate) *


(0:63992) CMND>>
```

8.2.3 Running a Simulation

After the essential OPTIONS are defined and any data collection has been set up the user may start the simulation by issuing a **run** COMMAND. At this point BISHOP allocates memory and initializes, the state space for the CA skeleton agent model, on the CM-2 computer. BISHOP then enters simulation mode. In this mode the simulation proceeds until the number of periods specified by the **MAXITER** OPTION is completed. Control is passed from the CI to the CA skeleton. The user may pause the simulation, and get back to command mode, by pressing any key. When a period of simulation completes the CA skeleton scans the keyboard and if a keypress has occurred the simulation is interrupted. The user may make changes to inessential OPTIONS or setup FILES and VARIABLES for data collection. The simulation continues when a **run** or **cont** COMMAND is issued. When the simulation has run for the specified number of periods BISHOP again enters command mode.

When the simulation is finished users should issue a **cfile** COMMAND to close each FILE that was opened. This will also deactivate and disassociate any VARIABLES that were associated to the FILE. The user can modify parameters and run another simulation or exit BISHOP ending the current session. The **quit** COMMAND is used to exit BISHOP. If essential OPTIONS need to be changed the user must issue a RESET COMMAND so the BISHOP can free memory resources on the CM-2 computer. This will cause BISHOP to reread the startup file and will leave BISHOP in the same state as when it was first executed. Changes to inessential OPTIONS do not require a reset.

Chapter 9

Discussion of Collected Data

The Hedonistic Dain Bramaged model was run, on the CM-2 computer at Thinking Machines Corporation, with a economy of 32^3 agents. There were 20 goods in the economy and the Market Omnipotence model of information propagation was used. Of course, one must remember that this model is not meant to be a valid representation of economic behavior. The graphs presented at the end of this chapter show the fluctuation in price for one good in this economy.

The periodic price fluctuations produced by this model are interesting because they represent the results of extreme irrationality in the behavior of individual agents. Consumers seek to buy from the merchant with the lowest price below their cutoff. They have perfect information on prices in the economy which results in every consumer trying to buy from the same merchant (the one with the lowest price). Consequently the merchant quickly sells what he planned to sell. All other consumers that come to him after that point are turned away. All of this produces an economy where a few consumers are successful in purchasing goods and only one merchant is successful each period. Merchants that fail to sell lower their prices by 10% each time. This produced the slow downward trend in average prices seen in figure 9-1. Merchant change to consumers, for the next period, when their price falls below 10.0. At the same time consumers who fail to purchase the desired good switch to merchants.

Figure 9-2 shows how many of the 32768 agents are merchants. In this model agents are either merchants or consumers therefore the graph of consumers is the inverse of figure 9-2. It is clear that the dominant factor in the periodicity of prices is the number of merchants in the economy. At some point a large number of merchants simultaneously switch to being consumers, as seen by the large dip in figure 9-2. These new consumers will almost all fail to buy during the next period and then switch back to being a merchant. The dips in figure 9-2 and the spikes in figure 9-1 coincide exactly. It is the sudden introduction of a large number of merchants, selling at relatively high prices, into the economy that produces the sudden jump in average price.

It might be interesting at some future date to run this model with initial conditions that endow agents with a large inventory and relatively small desire. In this way one merchant would not see out of his good as quickly and more consumers would be successful. This might change the dominating factor in price changes and produce completely different dynamic behavior. One might expect chaotic behavior to be observed for the right combination of initial conditions.

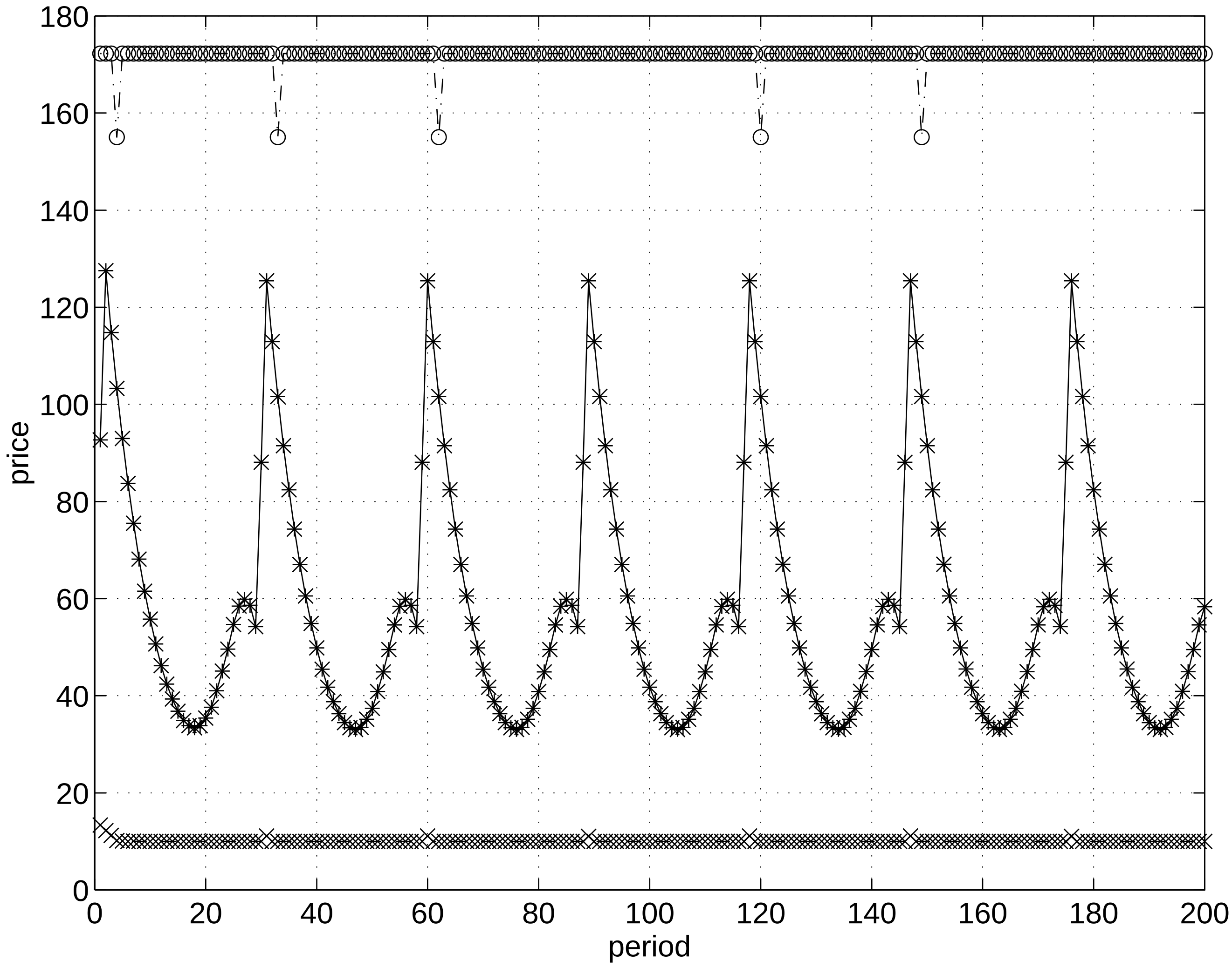

Figure 9-1: High, Average and Low Price

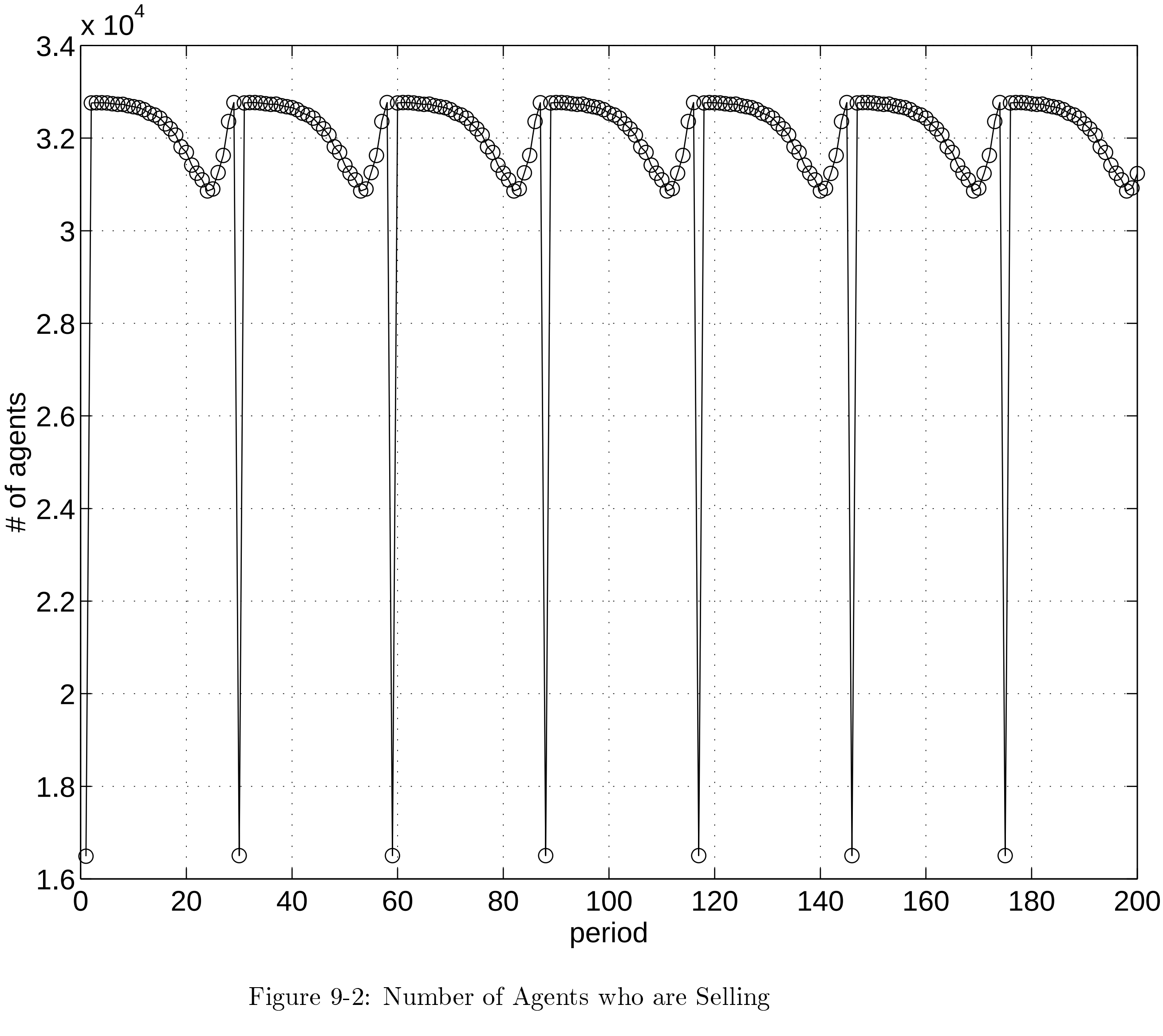

Figure 9-2: Number of Agents who are Selling

Appendix A

Code Specifications

A.1 BISHOP Flow Graph

Below is a rough flow graph of the major procedures in BISHOP.

```
main
   model_intro
   init_command
   init_bishop
   init_model
   init_util
   startup
   parse_cmndline
      run_bishop
         do_variable
         rank_goods
         diffuse_info
         update_diff_stats
         choose_merchant
         do_variable
         try_transaction
            update_sale_stats
         retry_transaction
         do_variable
         calculate_expectations
```

A.2 Data Structures

A.2.1 The CA State Space

```
extern shape [][][]lattice;

/* Transaction bookkeeping */
typedef struct trans {
  bool chng;
  unsigned char disp;
  unsigned char flag;
  int who;
  int quant;
  float price;
} TRANS;

extern float:lattice *money;
extern int:lattice *inv;
extern TRANS:lattice *plan;
extern TRANS:lattice *act;
```

A.2.2 The KEY Structure

```
typedef struct key_S {
  int c_addr;
  int rndc;
  int dist;
} KEY;
```

A.2.3 The BFILE Structure

```
typedef struct b_file {
  int cm;
  FILE *c, *s;
  int cnt, fi;
} BFILE;
```

A.2.4 The Command Interface Extensions Facility

```
typedef struct filespec {
  BFILE *fd;
```

```
  int flint;
} FILESPEC;

typedef struct optspec {
  char *value;
  int set;
  int vital;
} OPTSPEC;

typedef struct varspec {
  int fsc;
  int set;
} VARSPEC;

typedef struct breakspec {
  int set;
} BREAKSPEC;


typedef struct table {
  char *name;
  int empty;
  int type;
  void (*f)();
  char *data;
} TABLE;
```

A.3 Procedures

A.3.1 The CA Skeleton

Initialization and Cleanup

void `init_bishop`(int argc, char *argv[], int step)

In the first step initialize all internal BISHOP flags and parameters to their startup values. In the second step allocate memory for the CA state space. The command line arguments are passed from the `main` procedure for possible use.

void `finish_bishop`()

Deallocates memory used for the CA state space and initialize internal flags and parameters to their startup values.

State Space Updating and Simulation Stages

void `try_transaction`(int j)

All the logic of how transactions are conducted is contained in this procedure. The state space is updated here as well.

void `rank_goods`()

Rank goods by preference. Decide which goods to buy or sell.

void `diffuse_info`()

Propagate price information into the neighborhood.

void `update_diff_stats`()

Collect and update statistics about information received during the "Diffuse Information" stage.

void `choose_merchant`()

Choose a merchant to buy from for every good that is being bought this period.

void `retry_transaction`(int j)

Retry a failed transaction, for good j, with a modified plan in the hopes of have a partial success.

void `update_sale_stats`(int:lattice xm, int:lattice ym, int:lattice zm, int j)

Collect and update statistics about a successful transaction with the merchant at lattice location (xm, ym, zm) for good j.

void `calculate_expectations`()

Form and modify beliefs and expectations about the economy in the hopes of making better plans next period.

A.3.2 The Command Interface Extensions Facility

User Procedures

void `reg_spec`(TABLE *s, int code)

Register a list of extensions with the CI. Given a null terminated array of `TABLE` entries this procedure will copy the entries to an internal table. After extensions are registered they are available for use with the CI.

void `reg_one_spec`(TABLE t, int code)

Perform the same function as reg_spec but for one extension.

Core Procedures

int `parse_cmndline`()

Invoke the readline facility. Return -1 if any command that causes BISHOP to leave command mode is invoked and 0 otherwise.

int `parse_file`(char *filename)

Parse a file containing commands. The specified file is opened and each line is interpreted as if it were typed at the command prompt. It returns the same codes as `parse_cmndline`.

int `parse_fd`(FILE *fd)

Take a `FILE` pointer, for an already opened file, and perform the same actions as `parse_file`.

int `parse_string`(char *string)

Interprets string as a command line. It is the heart of the command interpreter. It returns the same codes as `parse_cmndline`.

void `startup`(int ignore_run)

Open the startup file, if present in the current directory, and invokes `parse_fd`. if ignore_run is 0 the **run** COMMAND is allowed in the startup file, otherwise the **run** COMMAND is ignored and causes file parsing to be aborted.

void `do_variable`(int code, int action, ...)

Perform the task indicated by action on the VARIABLE indicated by code. Other arguments may be given as needed by the particular task.

void `do_option`(int code, int action, ...)

Perform the task indicated by action on the OPTION indicated by code. Other arguments may be given as needed by the particular task.

void `do_breakpoint`(int code, int action, ...)

Perform the task indicated by action on the BREAKPOINT indicated by code. Other arguments may be given as needed by the particular task.

void `do_user`(int code, int action, ...)

Perform the task indicated by action on the user defined extension indicated by code. Other arguments may be given as needed by the particular task.

int `cont_safe`()

Check the status of all essential OPTIONS. Return 0 if any OPTION is unset and -1 otherwise. It is used to determine when it is safe to leave command mode and enter simulation mode.

void `print_help`(char flag, ...))

Print help information from either a string, file or via a function as specified by flag. If the type is string then the optional argument is a pointer to the string. If the type is file then the optional arguments are two pointers to the directory and filename of the file. If the type is function then the optional argument is a pointer to the function.

Utility Procedures

char **`argvize`(char *input, int *argc)

Break input into smaller pieces delimited by whitespace. Return a pointer to an array of strings in the style of argv. Write the size of this array to *argc.

int `find_in_cit`(char *name, int type)

Find extension that matches name and type, return the code that refers to the extension. It is primarily used by `parse_string` to lookup COMMANDS but may be used to find any type of extension.

`find_empty`(i)

This is macro that finds and empty entry in the command interface table. The index into the table is stored in i. It is used by `reg_spec` and `reg_one_spec` to insert new extensions in the table.

void `stripwhite`(char *string)

Strips the whitespace from the beginning and end of string.

A.3.3 The Command Interface Checkpointing Facility

void `sig_hdl`()

Signal handler to catch TERM and USR1 signals. Causes a checkpoint request so that BISHOP will be checkpointed at the next checkpoint call.

void `alrm_hdl`()

Signal handler for the ALRM signal. Used to initial one a time checkpoint at a certain time. Causes a checkpoint request and turns off the alarm.

A.3.4 The bstdio Library

BFILE *`bopen`(char *cmfn, char *sfn, char *mode, int fi)

Opens a BISHOP statistics file using cmfn and sfn as the filenames on the DataVault and front end respectively. Uses the same flags as `fopen` as specified by mode. Sets the file flush interval to fi.

void `bclose`(BFILE *file)

Closes and open BISHOP statistics file.

int `bprintf`(BFILE *file, char *format, ...)

Writes data, defined by format, to file. The optional arguments are the data to be written. Returns -1 if an error occurs and 0 otherwise. See code for explanation of format specifiers.

int `bscanf`(BFILE *file, char *format, int *sitm, int *pitm, ...)

Reads data, defined by format, from a file. The optional arguments are pointers to memory where the data to be read is stored. Stores the total number of items read from the front end and DataVault in *sitm and *pitm respectively. Returns -1 if an error occurs and 0 otherwise. See code for explanation of format specifiers.

A.4 CI Global Variables

The CI provides a set of global variables which are used to modify the behavior of the CI and provide information that is be needed by user defined procedures and the CA skeleton code. The user must take care in modifying these variables as incorrect or invalid values can cause BISHOP to crash or behave in an unpredictable manner. Some of these variables are flags which deactivate the specified feature, when they are 0, and activate the feature otherwise.

feprefix This is a string that contains the directory, on the front end, to which BISHOP is checkpointed. Users should not modify this directly but should construct a string containing the appropriate command and invoke the `parse_string` procedure.

cmprefix This is a string that contains the directory, on the DataVault, to which BISHOP is checkpointed. The same restriction that apply to feprefix apply to this variable.

cmnd_echo This flag controls whether or not commands read from files are echoed to the console. It may be safely modified at any time.

cmnd_log This flag controls whether of not commands interpreted by parse_string are written to a log file. The log file facility has not yet been implemented and this variable is not yet used.

ignore_run This flag determines whether or not the **run** COMMAND is ignored by the command interpreter. It may be modified at anytime but should be used with care as it will prevent BISHOP from entering simulation mode.

num_files This variable contains the total number of FILE extensions that have been registered with the CI. It is not to be changed by the user at any time.

num_cmnds This variable contains the total number of COMMAND extensions that have been registered with the CI. It is not to be changed by the user at any time.

num_vars This variable contains the total number of VARIABLE extensions that have been registered with the CI. It is not to be changed by the user at any time.

num_opts This variable contains the total number of OPTION extensions that have been registered with the CI. It is not to be changed by the user at any time.

num_breaks This variable contains the total number of BREAKPOINT extensions that have been registered with the CI. It is not to be changed by the user at any time.

CP This is an array of two entries that contain the periodic checkpoint interval. It should not be directly modified at any time by the user.

CA This variable specifies the time interval before an alarm signal is sent to the BISHOP process. It is part of the checkpoint facility and should not be directly modified by the user at any time.

Appendix B

The Code

B.1 bishop.hs

```
#ifndef BISHOP_HS_INCLUDED
#define BISHOP_HS_INCLUDED

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>

#include <cm/ckpt.h>

#include <default/bstdio.h>

#ifdef cstar
/* Bishop − A Framework for Economic Simulation Using Cellular Automata */

extern shape [][][]lattice;

/* Trasaction bookeeping */
typedef struct trans {
  bool chng;
  unsigned char disp;
  unsigned char flag;
  int who;
  int quant;
  float price;
} TRANS;

extern float:lattice *money;
extern int:lattice *inv;
extern TRANS:lattice *plan;
extern TRANS:lattice *act;
extern bool:lattice *hit;
#endif

extern int LS;
```

```
extern int G;
extern int maxiter;

extern int period;
extern int shapeset;
extern int defexts[];

#define EPBEG defexts[0]
#define EPLAN defexts[1]
#define EACT defexts[2]
#define EWLTH defexts[3]
#define EINV defexts[4]
#define EMONEY defexts[5]
#define EPEND defexts[6]
#define ELS defexts[7]
#define EG defexts[8]
#define EMAXITER defexts[9]
#define EFEP defexts[10]
#define ECMP defexts[11]
#define ECP defexts[12]
#define ECA defexts[13]


#ifdef cstar
#define SELF_ID (pcoord(0) + LS * pcoord(1) + LS * LS * pcoord(2))
#define ID(x, y, z) (x + LS * y + LS * LS * z)
#define MAPID(x, y, z) ((pcoord(0)+x)%%LS + LS * ((pcoord(1)+y)%%LS) + LS * LS * ((pcoord(2)+z)%%LS))
#define NMAP(v, xo, yo, zo) [(.+xo)%%LS][(.+yo)%%LS][(.+zo)%%LS]v
#define XS pcoord(0)
#define YS pcoord(1)
#define ZS pcoord(2)
#endif

#define NADA 'n'
#define HOLD 'h'
#define BUY 'b'
#define SELL 's'
#define HELD 'H'
#define BOUGHT 'B'
#define SOLD 'S'
#define FAIL 'F'

#define CUTOFF_LOW '<'
#define CUTOFF_HIGH '>'
#define HIGH_PRICES 'P'
#define NOT_SELLING '?'
#define LOW_INV 'I'
#define NO_ADS '*'
#define POVERTY '$'

#define PRELIM 0
#define FINAL 1

#define NO_ONE -1


void init_bishop();
void init_model();
void init_util();
```

```
void finish_bishop();
void finish_model();
void finish_util();
void try_transaction();
void choose_merchant();

/* Model hooks */
void (*diffuse_info)();
void (*rank_goods)(), (*update_diff_stats)();
void (*update_sale_stats)();
void (*calculate_expectations)();
void (*retry_transaction)();

void death();

void hdl_pbeg(), hdl_plan();
void hdl_act(), hdl_wlth();
void hdl_inv(), hdl_money();
void hdl_pend(), hdl_LS();
void hdl_G(), hdl_maxiter();
void hdl_FEP(), hdl_CMP();
void hdl_CA(), hdl_CP();

#endif
```

B.2 bishop.cs

```
#include <stdio.h>
#include <varargs.h>
#include <sys/types.h>
#include <sys/time.h>
#include <setjmp.h>

#include <pmem.hs>
#include <default/command.h>
#include <bishop.hs>
#include <model/model.hs>
#include <default/init_util.hs>



/* Bishop − A Cellular Automata Simulation of A Free Exchange Economy */

/* Flag specifying whether the shape is fully specified */
int shapeset = 0;

int period = 0;

/* Shape of the lattice */
shape [][][]lattice;


jmp_buf bishop_env;
```

```
int LS;
int G;
int maxiter;
float:lattice *money = (float:lattice *) NULL;
int:lattice *inv = (int:lattice *) NULL;
TRANS:lattice *plan = (TRANS:lattice *) NULL;
TRANS:lattice *act = (TRANS:lattice *) NULL;
bool:lattice *hit = (bool:lattice *) NULL;



main(int argc, char *argv[])
{
  OPTSPEC os;

  printf("BISHOP v5.0 Cellular Automata Simulation\n");
  printf("by:  Harris L. Gilliam (May 18, 1993)\n");
  model_intro();
  printf("\n\n\n");

  /* Init command interface */
  init_command(argc, argv);

  /* Do prelim init of bishop skeleton */
  init_bishop(argc, argv, PRELIM);

  /* Do prelim init of model */
  init_model(argc, argv, PRELIM);

  /* Do prelim init of utilities */
  init_util(argc, argv, PRELIM);

  /* Read commands in startup file */
  startup(0);

  /* This setjmp will be used by a longjmp in c_reset */
  setjmp(bishop_env);

  /* Parse commands until we have a lattice size */
  do {
    parse_cmndline();
    do_option(ELS, GETOPT, &os);
  } while(!os.set);

  with(lattice) {

    /* Parse commands until user quits */
    while(1) parse_cmndline();
  }
}



void run_bishop(int argc, char *argv)
{
  register int j;
  struct timeval timeout;
  fd_set fds;
```

```
char pm[64];

FD_ZERO(&fds);
FD_SET(fileno(stdin), &fds);
timeout.tv_sec = timeout.tv_usec = ((long) 0);


ignore_run = -1;

for (period = 0; period < maxiter; period++) {

  sprintf(pm, "Start Period %d", period);
  do_variable(ETIME, STATVAR, pm);

  if (select(ulimit(4,NULL), &fds, NULL, NULL, &timeout) > 0) {
    while(!parse_cmndline());
    FD_ZERO(&fds);
    FD_SET(fileno(stdin), &fds);
  }

  ckpt_periodic(feprefix, cmprefix);

  /* Write beginning of period marker */
  do_variable(EPBEG, STATVAR);

  /* Write out agents wealth */
  do_variable(EWLTH, STATVAR);


  /* All agents decide which goods to sell and which to buy */
  /* And then set the upper bounds and prices */
  if (rank_goods)
    rank_goods();

  ckpt_periodic(feprefix, cmprefix);

  /* All agents diffuse price information that they have.  For each */
  /* neighbor there is some probability that he will get this info */
  if (diffuse_info)
    diffuse_info();

  ckpt_periodic(feprefix, cmprefix);

  /* Update any statistics pertaining to diffused info */
  if (update_diff_stats)
    update_diff_stats();

  ckpt_periodic(feprefix, cmprefix);

  /* All agents decide who to buy from */
  choose_merchant();

  ckpt_periodic(feprefix, cmprefix);


  /* Write out agent's plan at */
  do_variable(EPLAN, STATVAR);
```

```
  /* All agents attempt to buy */
  for (j = 0; j < G; j++) {

    /* init all actual records */
    act[j].disp = NADA;
    act[j].flag = NADA;
    act[j].who = NO_ONE;
    act[j].quant = 0;
    act[j].price = 0.0F;


    where(plan[j].disp == BUY)
        if (CM_global_count_context() != 0) {

          try_transaction(j);

          ckpt_periodic(feprefix, cmprefix);

          if (retry_transaction)
            where ((plan[j].disp == BUY) &
                   (act[j].disp != BOUGHT))
              retry_transaction(j);

          ckpt_periodic(feprefix, cmprefix);
        }

    /* customers with no one to buy from fail */
    where((plan[j].disp == BUY) & (plan[j].who == NO_ONE)) {
        act[j].disp = FAIL;
        act[j].flag = plan[j].flag;
    }

    /* merchants who failed to sell this good */
    where ((plan[j].disp == SELL) & (act[j].disp == NADA)) {
        act[j].disp = FAIL;
        act[j].quant = 0;
    }

    /* By default all plans to hold succede */
    where (plan[j].disp == HOLD) {
        act[j].disp = HELD;
        act[j].quant = plan[j].quant;
    }
  } /* For (j = 0; */


  /* Write out agent's actuals at */
  do_variable(EACT, STATVAR);


  /* All agents calculate expectations for the next period */
  if (calculate_expectations)
    calculate_expectations();

  ckpt_periodic(feprefix, cmprefix);

  do_variable(EPEND, STATVAR);
} /* for(period = 0;*/
```

```
  ignore_run = 0;
  printf("\nSimulation has run for %d periods and is complete\n", period);
}
```

B.3 init_bishop.cs

```
#include <sys/signal.h>
#include <varargs.h>

#include <pmem.hs>
#include <default/rnd.hs>
#include <default/command.h>

#include <bishop.hs>

static int i1 = 0, i2 = 0;

static TABLE bishop_extensions[] = {
  { "PBEG", 0, VART, hdl_pbeg, NULL },
  { "PLAN", 0, VART, hdl_plan, NULL },
  { "ACT", 0, VART, hdl_act, NULL },
  { "WLTH", 0, VART, hdl_wlth, NULL },
  { "INV", 0, VART, hdl_inv, NULL },
  { "MONEY", 0, VART, hdl_money, NULL },
  { "PEND", 0, VART, hdl_pend, NULL },
  { "LS", 0, VART|OPTT, hdl_LS, NULL },
  { "G", 0, VART|OPTT, hdl_G, NULL },
  { "MAXITER", 0, VART|OPTT, hdl_maxiter, NULL },
  { "FEP", 0, VART|OPTT, hdl_FEP, NULL },
  { "CMP", 0, VART|OPTT, hdl_CMP, NULL },
  { "CP", 0, VART|OPTT, hdl_CP, NULL },
  { "CA", 0, VART|OPTT, hdl_CA, NULL },
  { NULL, NULL, NULL, NULL, NULL }
};


int defexts[15];

void sig_alrm();



void init_bishop(int argc, char *argv[], int step)
{
  register int i;


  switch (step) {

  case PRELIM: {

    if (!i1) {
      /* Register default extensions */
      reg_spec(bishop_extensions, defexts);
```

```
      /* Init all hooks to NULL */
      diffuse_info = update_diff_stats = rank_goods = update_sale_stats = NULL;
      calculate_expectations = retry_transaction = NULL;
      i1 = -1;
    }
    break;
  }

  case FINAL: {

    if (!i2) {
      srand(getuid() + time(NULL));

      hit = (bool:lattice *) PMalloc(boolsizeof(bool:lattice));

      money = (float:lattice *) PMalloc(boolsizeof(float:lattice));
      inv = (int:lattice *) PMalloc(G * boolsizeof(int:lattice));
      plan = (TRANS:lattice *) PMalloc(G * boolsizeof(TRANS:lattice));
      act = (TRANS:lattice *) PMalloc(G * boolsizeof(TRANS:lattice));

      /* Initialize each agents wealth and init plan */
      for (i = 0; i < G; i++) {
            plan[i].chng = 0;
            plan[i].flag = NADA;
            plan[i].who = NO_ONE;
            inv[i] = rnd(10) * 100 + rnd(200);
      }

      (*money) = frnd(75000.0F) + frnd(1000.0F);
      i2 = -1;

      /* Write bishop parameters that don't change */
      do_variable(ELS, STATVAR);
      do_variable(EG, STATVAR);
      do_variable(EMAXITER, STATVAR);
    }
    break;
  }
  }
}


void finish_bishop()
{
  if (i2) {
    G = LS = maxiter = 0;

    PFree(money);
    PFree(inv);
    PFree(plan);
    PFree(act);
    PFree(hit);

    deallocate_shape(&lattice);

  }
  i2 = 0;
}
```

```
/*************************************************************************/


void hdl_LS(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;


  switch(type) {

  case OPTT: {
    switch (action) {

    case SETOPT: { /* write to option variable */
      int ac;
      char **av;;

      next_arg(ac, args, int);
      next_arg(av, args, char **);

      if (ac == 1) {
            LS = atoi(av[0]);
            oset = -1;

            /* Allocate three-d lattice */
            if (!shapeset) {
              lattice = allocate_shape(&lattice, 3, LS, LS, LS);
              CM_set_vp_set(lattice);
              shapeset = -1;
            }
      }

      break;
    }

    case PRTOPT: { /* print option variable */

      if (oset)
            printf("%d (lattice size) *\n", LS);
      else
            printf("unset (lattice size) *\n");

      break;
    }

    case GETOPT: {
      OPTSPEC *os;

      next_arg(os, args, OPTSPEC *);
      os->value = (char *) &LS;
      os->set = oset;
      os->vital = -1;
      break;
    }
```

```
    case OPTVITAL: {
      int *safe;

      next_arg(safe, args, int *);
      *safe &= oset;
      break;
    }
    }
    break;
  }

  case VART: {
    switch(action) {

    case STATVAR: {
      FILESPEC *fs;

      if (vset) {
            fs = (FILESPEC *) cit[fsc].data;

            if (bprintf(fs->fd, "%K%D\n", "LS", LS))
              printf("write of LS var failed");
      }
      break;
    }

    case GETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      vs->fsc = fsc;
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}


void hdl_G(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;
```

```
switch(type) {

case OPTT: {
  switch (action) {

  case SETOPT: { /* write to option variable */
    int ac;
    char **av;;

    next_arg(ac, args, int);
    next_arg(av, args, char **);

    if (ac == 1) {
          G = atoi(av[0]);
          oset = -1;
    }
    break;
  }

case PRTOPT: { /* print option variable */

  if (oset)
    printf("%d (# goods in economy) *\n", G);

  else
    printf("unset (# goods in economy) *\n");

  break;
}

case GETOPT: { /* store value in memory pointed to by av */
  OPTSPEC *os;

  next_arg(os, args, OPTSPEC *);
  os->value = (char *) &G;
  os->set = oset;
  os->vital = -1;
  break;
}

  case OPTVITAL: {
    int *safe;

    next_arg(safe, args, int *);

    *safe &= oset;
    break;
  }
  }
  break;
}

case VART: {
  switch(action) {

  case STATVAR: {
    FILESPEC *fs;
```

```
    if (vset) {
        fs = (FILESPEC *) cit[fsc].data;

        if (bprintf(fs->fd, "%K%D\n", "G", G))
          printf("write of G var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = vset;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
        fsc = vs->fsc;

    if (vs->set != NA)
        vset = vs->set;
    break;
  }
  }
  break;
 }
 }
}



void hdl_maxiter(int type, int action, char *args)
{
 static int oset = 0, vset = 0;
 static int fsc = -1;



 switch(type) {

 case OPTT: {
  switch (action) {

  case SETOPT: { /* write to option variable */
    int ac;
    char **av;;

    next_arg(ac, args, int);
    next_arg(av, args, char **);
```

```
    if (ac == 1) {
        maxiter = atoi(av[0]);
        oset = -1;
    }
    break;
  }

  case PRTOPT: { /* print option variable */

    if (oset)
        printf("%d (# of periods to run simulation) *\n", maxiter);

    else
        printf("unset (# of periods to run simulation) *\n");

    break;
  }

  case GETOPT: {
    OPTSPEC *os;

    next_arg(os, args, OPTSPEC *);
    os->value = (char *) &maxiter;
    os->set = oset;
    os->vital = -1;
    break;
  }
  }
  break;
}

case VART: {
  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (vset) {
        fs = (FILESPEC *) cit[fsc].data;

        if (bprintf(fs->fd, "%K%D\n", "MAXITER", maxiter))
          printf("write of maxiter var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = vset;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;
```

```
      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}



void hdl_FEP(int type, int action, char *args)
{
  static int oset = -1, vset = 0;
  static int fsc = -1;

  switch(type) {
  case OPTT: {

    switch (action) {

    case SETOPT: { /* write to option variable */
      int ac;
      char **av;;

      next_arg(ac, args, int);
      next_arg(av, args, char **);

      if (ac == 1) {
            strcpy(feprefix, av[1]);
            oset = -1;
      }
      break;
    }

    case PRTOPT: { /* print option variable */

      if (oset)
            printf("%s (front end checkpoint prefix) *\n", feprefix);

      else
            printf("unset (front end checkpoint prefix) *\n");

      break;
    }

    case GETOPT: {
      OPTSPEC *os;

      next_arg(os, args, OPTSPEC *);

      os->value = feprefix;
      os->set = oset;
      os->vital = -1;
```

```
    }
    }
    break;
  }

  case VART: {
    switch(action) {

    case STATVAR: {
      FILESPEC *fs;

      if (vset) {
            fs = (FILESPEC *) cit[fsc].data;

            if (bprintf(fs->fd, "%K%S\n", "FEP", feprefix))
              printf("write of FEP var failed\n");
      }
      break;
    }

    case GETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);

      vs->fsc = fsc;
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}


void hdl_CMP(int type, int action, char *args)
{
  static int oset = -1, vset = 0;
  static int fsc = -1;

  switch(type) {
  case OPTT: {

    switch (action) {
```

```
  case SETOPT: { /* write to option variable */
    int ac;
    char **av;;

    next_arg(ac, args, int);
    next_arg(av, args, char **);

    if (ac == 1) {
          strcpy(cmprefix, av[1]);
          oset = -1;
    }
    break;
  }

  case PRTOPT: { /* print option variable */

    if (oset)
          printf("%s (CM checkpoint prefix) *\n", cmprefix);

    else
          printf("unset (CM checkpoint prefix) *\n");

    break;
  }

  case GETOPT: {
    OPTSPEC *os;

    next_arg(os, args, OPTSPEC *);

    os->value = cmprefix;
    os->set = oset;
    os->vital = -1;
  }
  }
  break;
}

case VART: {
  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (vset) {
          fs = (FILESPEC *) cit[fsc].data;

          if (bprintf(fs->fd, "%K%S\n", "CMP", cmprefix))
            printf("write of CMP var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
```

```
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}


void hdl_CP(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;
  static char a[3][3] = { "opt", "CA", "0" };


  switch(type) {

  case OPTT: {
    switch (action) {

    case SETOPT: { /* write to option variable */
      int ac;
      char **av;;

      next_arg(ac, args, int);
      next_arg(av, args, char **);

      if (ac == 2) {
            CP[0] = atoi(av[0]);
            CP[1] = atoi(av[1]);

            if (CP[0] == 0 && CP[1] == 0) {
              oset = 0;
              ckpt_periodic_end();
            }
            else {
              oset = -1;
              c_opt(CMNDT, XCMND, 3, a);
              signal(SIGALRM, SIG_DFL);
              ckpt_periodic_start(CP[0], CP[1]);
            }
      }
      break;
    }
```

```
  case PRTOPT: { /* print option variable */

    if (oset)
         printf("%d:%d (periodic checkpoint time)\n", CP[0], CP[1]);

    else
         printf("unset (periodic checkpoint time)\n");

    break;
  }

  case GETOPT: {
    OPTSPEC *os;

    next_arg(os, args, OPTSPEC *);
    os->value = (char *) CP;
    os->set = oset;
    os->vital = 0;
    break;
  }
  }
  break;
}

case VART: {
  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (vset) {
         fs = (FILESPEC *) cit[fsc].data;

         if (bprintf(fs->fd, "%K%D%D", "CP", CP[0], CP[1]))
           printf("write of CP var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = vset;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
         fsc = vs->fsc;

    if (vs->set != NA)
         vset = vs->set;
    break;
```

```
    }
    }
    break;
  }
  }
}

void hdl_CA(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;
  static char a[4][3] = { "opt", "CP", "0", "0" };

  switch(type) {

  case OPTT: {
    switch (action) {

    case SETOPT: { /* write to option variable */
      int ac;
      char **av;;

      next_arg(ac, args, int);
      next_arg(av, args, char **);

      if (ac == 1) {
            CA = atoi(av[0]);

            if (CA == 0) {
              oset = 0;
              alarm(0);
              signal(SIGALRM, SIG_DFL);
            }
            else {
              oset = -1;
              ckpt_periodic_end();
              c_opt(CMNDT, XCMND, 4, a);
              signal(SIGALRM, sig_alrm);
              alarm(CA);
            }
      }
      break;
    }

    case PRTOPT: { /* print option variable */

      if (oset)
            printf("%d (alarm checkpoint time)\n", CA);

      else
            printf("unset (alarm checkpoint time)\n");

      break;
    }

    case GETOPT: {
```

```
      OPTSPEC *os;

      next_arg(os, args, OPTSPEC *);
      os->value = (char *) &CA;
      os->set = oset;
      os->vital = 0;
      break;
    }
    }
    break;
  }

  case VART: {
    switch(action) {

    case STATVAR: {
      FILESPEC *fs;

      if (vset) {
            fs = (FILESPEC *) cit[fsc].data;

            if (bprintf(fs->fd, "%K%D\n", "CA", CA))
              printf("write of CA var failed\n");
      }
      break;
    }

    case GETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);

      vs->fsc = fsc;
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}


void hdl_plan(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static int set = 0;
```

```
static int fsc = -1;
static char fn[256];

switch(action) {

case ACTVAR: {
  if (!ofd) {
    sprintf(fn, "%s_%s.out", cit[fsc].name, "PLAN");
    if ((ofd = fopen(fn, "w")) == NULL)
          printf("\n\nhdl_plan:  Couldn't open ouput file %s\n\n", fn);
  }
  else {
    if ((ofd = fopen(fn, "a+")) == NULL)
          fprintf(stderr, "\n\nhdl_plan:  Couldn't open ouput file %s\n\n", fn);
  }
  break;
}

case DACTVAR: {
  if (ofd)
    fclose(ofd);
  break;
}

case STATVAR: {
  register int i, j;
  FILESPEC *fs;

  if (set) {
    fs = (FILESPEC *) cit[fsc].data;

    if (bprintf(fs->fd, "%K%b[]", "PLAN",
                    G * boolsizeof(TRANS:lattice), plan))
          printf("write of PLAN var failed\n");

  /* Write data */
    for(j = 0; j < G; j++) {

          /* CUTOFF low, ave, high, # */
          where(plan[j].disp == BUY) {
            i = CM_global_count_context();
            if (i != 0) {
              fprintf(ofd, "%10.5f %10.5f %10.5f %d ",
                        <?= plan[j].price,
                        (+= plan[j].price) / (float ) i,
                        >?= plan[j].price,
                        i);
            }
            else
              fprintf(ofd, "0.0 0.0 0.0 0 ");
          }

          /* PRICE low, ave, high, # */
          where(plan[j].disp == SELL) {
            i = CM_global_count_context();
            if (i != 0) {
              fprintf(ofd, "%10.5f %10.5f %10.5f %d ",
                        <?= plan[j].price,
```

```
                    (+= plan[j].price) / (float ) i,
                    >?= plan[j].price,
                    i);
            }
            else
              fprintf(ofd, "0.0 0.0 0.0 0 ");
          }
      }
      fprintf(ofd, "\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}



void hdl_act(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static int set = 0;
  static int fsc = -1;
  static char fn[256];


  /* open a output file if one is not open */


  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "ACT");
      if ((ofd = fopen(fn, "w")) == NULL)
           printf("\n\nhdl_act:  Couldn't open ouput file %s\n\n", fn);
    }
    else {
```

```
    if ((ofd = fopen(fn, "a+")) == NULL)
        fprintf(stderr, "\n\nhdl_act:  Couldn't open ouput file %s\n\n", fn);
  }
  break;
}

case DACTVAR: {
  if (ofd)
    fclose(ofd);
  break;
}

case STATVAR: {
  register int i, j;
  FILESPEC *fs;

  if (set) {
    fs = (FILESPEC *)cit[fsc].data;

    if (bprintf(fs->fd, "%K%b[]", "ACT", G * boolsizeof(TRANS:lattice), act))
        printf("write of ACT var failed\n");

    /* Write data */
    for(j = 0; j < G; j++) {

        /* PURCHASES low, ave, high, # */
        where(act[j].disp == BOUGHT) {
          i = CM_global_count_context();
          if (i != 0) {
            fprintf(ofd, "%10.5f %10.5f %10.5f %d ",
                    <?= act[j].price,
                    (+= act[j].price) / (float) i,
                    >?= act[j].price,
                    i);
          }
          else
            fprintf(ofd, "0.0 0.0 0.0 0 ");
        }

        /* SALES low, ave, high, # */
        where(act[j].disp == SOLD) {
          i = CM_global_count_context();
          if (i != 0) {
            fprintf(ofd, "%10.5f %10.5f %10.5f %d ",
                    <?= act[j].price,
                    (+= act[j].price) / (float) i,
                    >?= act[j].price,
                    i);
          }
          else
            fprintf(ofd, "0.0 0.0 0.0 0 ");
        }
    }
    fprintf(ofd, "\n");
  }
  break;
}

case GETVAR: {
```

```
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}



void hdl_wlth(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static char fn[256];
  static int set;
  static int fsc = -1;



  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "WLTH");
      if ((ofd = fopen(fn, "w")) == NULL)
           printf("\n\nhdl_wlth:  Couldn't open ouput file %s\n\n", fn);
    }
    else {
      if ((ofd = fopen(fn, "a+")) == NULL)
           fprintf(stderr, "\n\nhdl_wlth:  Couldn't open ouput file %s\n\n", fn);
    }
    break;
  }

  case DACTVAR: {
    if (ofd)
      fclose(ofd);
    break;
  }

  case STATVAR: {
    register int j;
    FILESPEC *fs;
```

```
    if (set) {
      fs = (FILESPEC *)cit[fsc].data;

      bprintf(fs->fd, "%K%f%d[]", "WLTH", money, G, inv);

      fprintf(ofd, "%10.5f %10.5f ", <?= (*money), >?= (*money));

      for(j = 0; j < G; j++)
            fprintf(ofd, "%d %d ", <?= inv[j], >?= inv[j]);

      fprintf(ofd, "\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}



void hdl_inv(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
      FILESPEC *fs;

      if (set) {
            fs = (FILESPEC *) cit[fsc].data;

            if (bprintf(fs->fd, "%K%d[]", "INV", G, inv))
              printf("write of INV var failed\n");
      }
    break;
```

```
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}



void hdl_money(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      if (bprintf(fs->fd, "%K%f", "MONEY", money))
            printf("write of MONEY var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }
```

```
  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}



void hdl_pbeg(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      if (bprintf(fs->fd, "%K%D\n", "PBEG", period))
           printf("write of PBEG var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}
```

```
void hdl_pend(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      if (bprintf(fs->fd, "%K", "PEND"))
           printf("write of PEND var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}
```

B.4 try_transaction.cs

```
#include <stdlib.h>

#include <cm/paris.h>

#include <bishop.hs>
#include <default/gcoord.hs>
```

```
typedef struct key_S {
  int c_addr;
  int rndc;
  int dist;
} KEY;


void try_transaction(int j)
{
  int:lattice xi, yi, zi;
  bool:lattice done = 0;
  KEY:lattice key;
  KEY:lattice pick;
  int:lattice sa;


  where(plan[j].who != NO_ONE)
    if (CM_global_count_context() != 0) {

    g_coord(plan[j].who, &xi, &yi, &zi, LS);
    key.dist = (abs(xi) > abs(yi) ?
                    (abs(xi) > abs(zi) ? abs(xi) : abs(zi)) :
                    (abs(yi) > abs(zi) ? abs(yi) : abs(zi)));
    key.rndc = prand();
    key.c_addr = SELF_ID;


    /* Make send address from news coordinates */
    CM_make_news_coordinate_1L(CM_vp_set_geometry(lattice),
                               &sa, 0, &xi, 32);
    CM_deposit_news_coordinate_1L(CM_vp_set_geometry(lattice),
                                  &sa, 1, &yi, 32);
    CM_deposit_news_coordinate_1L(CM_vp_set_geometry(lattice),
                                  &sa, 2, &zi, 32);

    while ((&= done) == 0) {

      /* Init pick to a value larger that any customers distance */
      everywhere
            pick.dist = LS + 1;

      where (done == 0) {
            /* Send with min */
            CM_send_with_u_min_1L(&pick, &sa, &key, boolsizeof(key), CM_no_field);

            where([xi][yi][zi]pick.c_addr == SELF_ID) {

              /* Make sure this merchant is selling this good and          */
              /* this merchant is selling at least as much as I want and */
              /* the merchant's price fits (<=) my cuttoff and             */
              /* I have enough money to buy                                */

              /* If my cutoff is high, make a note and buy at the lower price */
              where([xi][yi][zi]plan[j].price < plan[j].price) {
                act[j].flag = CUTOFF_HIGH;
                plan[j].price = [xi][yi][zi]plan[j].price;
              }
```

```
        /* If this guy isn't selling I screwed up in choose_merchant */
        where([xi][yi][zi]plan[j].disp != SELL) {
          act[j].disp = FAIL;
          act[j].flag = NOT_SELLING;
        }

        /* If he doesn't have at least as much as I want then make a note */
        /* I should have checked in choose_merchant, or maybe someone else */
        /* bought all of it up before me */
        where([xi][yi][zi]plan[j].quant < plan[j].quant) {
          act[j].disp = FAIL;
          act[j].flag = LOW_INV;
          [xi][yi][zi]act[j].flag = LOW_INV;
        }

        /* If his price is too high then I screwed up in choose_merchant */
        where([xi][yi][zi]plan[j].price > plan[j].price) {
          act[j].disp = FAIL;
          act[j].flag = CUTOFF_LOW;
          [xi][yi][zi]act[j].flag = CUTOFF_LOW;
        }

        /* If I don't have the cashola then I screwed up in rank_goods */
        where([xi][yi][zi]plan[j].price * plan[j].quant >
                *money) {
          act[j].disp = FAIL;
          act[j].flag = POVERTY;
          [xi][yi][zi]act[j].flag = POVERTY;
        }

        /* I'm ok !! I can purchase !! */
        where(act[j].disp == NADA) {
          /* commence transaction */

          /* record transaction in merchant's records */
          [xi][yi][zi]act[j].who = SELF_ID;
          [xi][yi][zi]act[j].quant += plan[j].quant;
          [xi][yi][zi]act[j].price = plan[j].price;
          [xi][yi][zi]act[j].disp = SOLD;

          /* transaction in customer's records */
          act[j].who = plan[j].who;
          act[j].quant = plan[j].quant;
          act[j].price = plan[j].price;
          act[j].disp = BOUGHT;

          /* merchant's inventory decreases */
          [xi][yi][zi]inv[j] -= plan[j].quant;

#ifndef SELLALL
          /* merchant's  planned sales decreases */
          [xi][yi][zi]plan[j].quant -= plan[j].quant;
#endif

          /* customer's inventory increases */
          inv[j] += plan[j].quant;

          /* merchant's money increases */
```

```
            [xi][yi][zi](*money) += plan[j].quant * plan[j].price;

            /* customer's money decreases */
            (*money) -= plan[j].quant * plan[j].price;

            if (update_sale_stats)
              update_sale_stats(xi, yi, zi, j);
          } /* where(act[j].disp == NADA) */

          /* These customers are done */
          done = 1;

        } /* where([xi][yi][zi]pick.c_addr == SELF_ID) */
      } /* where (done == 0) */
    } /* while ((&= done) == 0) */
  } /* if (CM_global_count_context() != 0) */
}
```

B.5 command.h

```
#ifndef COMMAND_HS_INCLUDED
#define COMMAND_HS_INCLUDED


#include <cm/paris.h>
#include <default/bstdio.h>



typedef struct filespec {
  BFILE *fd;
  int flint;
} FILESPEC;

typedef struct optspec {
  char *value;
  int set;
  int vital;
} OPTSPEC;

typedef struct varspec {
  int fsc;
  int set;
} VARSPEC;

typedef struct breakspec {
  int set;
} BREAKSPEC;


typedef struct table {
  char *name;
  int empty;
  int type;
  void (*f)();
```

```
  char *data;
} TABLE;


#define CITSZ 2048
#define CMNDT 0x1
#define VART 0x2
#define FILET 0x4
#define BREAKT 0x8
#define OPTT 0x10
#define USERT 0x2000


/* Actions */
#define SETOPT 100
#define GETOPT 110
#define PRTOPT 120
#define OPTVITAL 130

#define ACTVAR 200
#define DACTVAR 210
#define WVAR 220
#define RVAR 230
#define STATVAR 240
#define GETVAR 250
#define SETVAR 260

#define FOPEN 300
#define FCLOSE 310
#define FFLUSH 320

#define XCMND 400

#define BRKPNT 500
#define GETBRK 510
#define SETBRK 520

#define HELP 600

#define XUSER 700
#define GETUSR 710

#define NA -1000

/* Types of help */
#define FILE_HELP 'f'
#define STRING_HELP 's'
#define FUNC_HELP 'F'


#define find_empty(i) \
{ for(i = 0; i < CITSZ; i++) \
    if (cit[i].empty) \
      break; \
  if (i == CITSZ) { \
    fprintf(stderr, "\nFATAL: Command interface table has overflown\n"); \
    death(0); \
```

```
  } \
}

#define next_arg(v, a, t) \
{ \
  (v) = ((t *) (a))[0]; \
  (a) += sizeof(t); \
}


extern char feprefix[50], cmprefix[50];
extern int CP[2], CA;
extern int cmnd_echo, cmnd_log;
extern TABLE cit[CITSZ];
extern int num_vars, num_files, num_cmnds, num_breaks, num_opts;
extern int ignore_run;

void stripwhite();

void startup();
int parse_file();
int parse_fd();
int parse_string();
int parse_cmndline();
void reg_spec();
void reg_one_spec();
void do_variable();
void do_option();
void do_breakpoint();
void do_user();
int find_in_cit();
char **argvize();
int cont_safe();
void print_help();

/***********************************************************************/

void c_file(), c_cfile();
void c_assoc(), c_source();
void c_check(), c_break();
void c_opt(), c_log();
void c_echo(), c_help();
void c_dassoc(), c_run();
void c_quit(), c_reset();
void c_load(), c_save();
void c_var(), c_cont();

void sig_hdl();
void alrm_hdl();

#endif
```

B.6 command.c

```
#include <stdio.h>
#include <string.h>
#include <rl/readline.h>
#include <rl/history.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/signal.h>
#include <varargs.h>
#include <setjmp.h>

#include <mem.h>
#include <default/command.h>

#include <bishop.hs>


extern CM_vp_set_id_t lattice;
extern jmp_buf bishop_env;


static char start_file[128];
static int ac_save;
static char **av_save;
static int iflag = 0;

static TABLE commands[] = {
  { "run", 0, CMNDT, c_run, NULL },
  { "cont", 0, CMNDT, c_cont, NULL },
  { "quit", 0, CMNDT, c_quit, NULL },
  { "reset", 0, CMNDT, c_reset, NULL },
  { "file", 0, CMNDT, c_file, NULL },
  { "cfile", 0, CMNDT, c_cfile, NULL },
  { "assoc", 0, CMNDT, c_assoc, NULL },
  { "dassoc", 0, CMNDT, c_dassoc, NULL },
  { "var", 0, CMNDT, c_var, NULL },
  { "source", 0, CMNDT, c_source, NULL },
  { "check", 0, CMNDT, c_check, NULL },
  { "break", 0, CMNDT, c_break, NULL },
  { "opt", 0, CMNDT, c_opt, NULL },
  { "log", 0, CMNDT, c_log, NULL },
  { "echo", 0, CMNDT, c_echo, NULL },
  { "help", 0, CMNDT, c_help, NULL },
  { "load", 0, CMNDT, c_load, NULL },
  { "save", 0, CMNDT, c_save, NULL },
  { NULL, NULL, NULL, NULL, NULL }
};


TABLE cit[CITSZ];
int num_vars = 0, num_files = 0, num_cmnds = 0, num_breaks = 0, num_opts = 0;

char feprefix[50], cmprefix[50];
int CP[2] = {0, 0};
int CA = 0;
```

```
int cmnd_echo = 1;
int cmnd_log = 0;
int ignore_run = 0;
int exit_cmnd_mode = 0;
int running = 0;

/* Log file */
FILE *logfile = NULL;


void init_command(ac, av)
    int ac;
    char *av[];
{
  register int i;


  if (!iflag) {

  /* default startup filename */
    strcpy(start_file, "bishop.startup");

    if (ac != 1) {
      for (i=1; i<ac; i++) {

            if (strncmp(av[i], "-F", 2) == 0) {
              strcpy(start_file, av[i]+2);
            }
      }
    }

    /* init tables */
    num_vars = num_files = num_cmnds = num_breaks = num_opts = 0;

    /* Save argc and argv */
    ac_save = ac;
    av_save = av;

    /* Checkpoint prefix */
    sprintf(feprefix, "%s/checkpoints/", FEHOME);
    sprintf(cmprefix, "%s/checkpoints/", DVHOME);

    /* Init checkpointing package */
    ckpt_init(av[0]);
    ckpt_verbose = 1;
    signal(SIGTERM, sig_hdl);
    signal(SIGUSR1, sig_hdl);

    /* Mark all table entries as empty */
    for(i = 0; i < CITSZ; i++)
      cit[i].empty = -1;

    /* Register the default commands */
    reg_spec(commands, NULL);

    iflag = -1;
  }
}
```

```
void finish_command()
{
  register int i;
  static char *a[2] = { "cfile", NULL };

  if (iflag) {

    /* Periodic checkpoint time */
    CP[0] = CP[1] = 0;

    /* Alarm checkpoint time */
    CA = 0;

    for(i = 0; i < CITSZ; i++) {

      if (!cit[i].empty) {

          switch (cit[i].type & FILET) {

          case FILET: {
            a[1] = cit[i].name;
            c_cfile(CMNDT, XCMND, 2, a);
            break;
          }
          }
      }
      cit[i].empty = -1;
    }

    num_cmnds = num_vars = num_files = num_breaks = num_opts = 0;
    iflag = 0;
  }
}



void startup(irun)
     int irun;
{
  struct stat b;
  int irun_save = ignore_run;
  FILE *fd;


  /* Set run ignore flag */
  ignore_run = irun;

  /* Read in commands from startup file (if it exists) */
  if (!stat(start_file, &b)) {
    if ((fd = fopen(start_file, "r")) == NULL)
      fprintf(stderr, "Can't open startup file:  %s\n", start_file);

    else {
      printf("\nReading commands in startup file:  %s\n\n", start_file);
      parse_fd(fd);
      fclose(fd);
```

```
    }
  }

  c_opt(CMNDT, XCMND, 1, NULL);
  ignore_run = irun_save;
}


void reg_spec(s, code)
     TABLE *s;
     int *code;
{
  static char zero[128] = "\0";
  register int i;

  for(i = 0; bcmp(s+i, zero, sizeof(TABLE)); i++) {
    reg_one_spec(s[i], code);
    if (code)
      code++;
  }
}


void reg_one_spec(t, code)
     TABLE t;
     int *code;
{
  register int i;

  find_empty(i);

  bcopy(&t, cit + i, sizeof(TABLE));
  cit[i].empty = 0;

  if ((t.type & CMNDT) == CMNDT)
    num_cmnds++;

  if ((t.type & VART) == VART)
    num_vars++;

  if ((t.type & BREAKT) == BREAKT)
    num_breaks++;

  if ((t.type & OPTT) == OPTT)
    num_opts++;

  if (code)
    *code = i;
}


void stripwhite(t)
     char *t;
```

```
{
  register int i = 0;


  while(whitespace(t[i]))
    i++;

  if (i)
    strcpy(t, t + i);

  i = strlen(t) - 1;

  while(i > 0 && whitespace(t[i]))
    i--;

  t[++i] = '\0';
}



int parse_file(fn)
     char *fn;
{
  FILE *fd;
  int i;


  if ((fd = fopen(fn, "r")) == NULL) {
    printf("\nCan't open file:  %s\n", fn);
    return(0);
  }
  else {
    i = parse_fd(fd);
    fclose(fd);
    return(i);
  }
}



int parse_fd(fd)
     FILE *fd;
{
  char line[512];


  while(!feof(fd)) {

    line[0] = '\0';
    fgets(line, 512, fd);

    if (feof(fd))
      return(0);

    if (cmnd_echo)
      printf("%s\n", line);
```

```
    if (parse_string(line))
      return (-1);
  }
  return(0);
}



int parse_string(l)
     char *l;
{
  char **av;
  int ac;
  int i;
  char cmnd[10];


  if (cmnd_log && logfile)
    fprintf(logfile, "CMND>> %s\n", l);

  sscanf(l, "%s", cmnd);

  /* Find command in table */
  if ((i = find_in_cit(cmnd, CMNDT)) == -1) {
    printf("\n\nunrecognized command:  %s\n\t%s\n\n", cmnd, l);
    return(0);
  }

  else {
    if (cit[i].f) {
      av = argvize(l, &ac);
      (*(cit[i].f))(CMNDT, XCMND, ac, av);
      free(av);
      printf("\n");
    }

    else
      printf("\n\nNull command\n\n");

    return(exit_cmnd_mode);
  }
}



int parse_cmndline()
{
  register char *line;
  char prompt[64];


  sprintf(prompt, "\n(%d:%d) CMND>> ", period, CM_available_memory());
  line = readline(prompt);

  if (line) {
    stripwhite(line);

    if (*line) {
```

```
      add_history(line);
      if (parse_string(line)) {
            free(line);
            return(-1);
      }

      free(line);
    }
  }
  return(0);
}



int cont_safe()
{
  register int i;
  int safe = -1;

  for (i = 0; i < CITSZ; i++) {
    if ((!cit[i].empty) && ((cit[i].type & OPTT) == OPTT))
      do_option(i, OPTVITAL, &safe);
  }
  return(safe);
}



int find_in_cit(n, type)
     char *n;
     int type;
{
  register int i;

  for (i = 0; i < CITSZ; i++)
    if ((!cit[i].empty) &&
            ((cit[i].type & type) == type) &&
            (!strcmp(cit[i].name, n)))
      return(i);

  return(-1);
}



int check_args(fn, m, n, ac)
     char *fn;
     int m;
     int n;
     int ac;
{
  if (n == 0) { /* no args so notify user that they args are ignored */
    if (ac != 1)
      printf("\n%s:  arguments ignored\n", fn);
    return(-1);
  }
```

```
  if (m && (ac != n)) { /* need exactly n-1 args */
    printf("\n%s:  need exactly %d arguements\n", fn, n-1);
    return(0);
  }

  if (!m && (ac < n)) { /* need at least n-1 args */
    printf("\n%s:  need at least %d arguements\n", fn, n-1);
    return(0);
  }

  return(-1);
}



void print_help(type, dir, file)
     char type;
     char *dir;
     char *file;
{
  char cmd[1024];
  void (*f)();

  switch(type) {

  case FILE_HELP: {
    sprintf(cmd, "cat %s/%s", dir, file);
    system(cmd);
    break;
  }

  case STRING_HELP: {
    printf("%s\n", dir);
    break;
  }

  case FUNC_HELP: {
    f = (void (*)()) dir;
    (*f)();
    break;
  }

  default: {
    printf("print_help:  can't provide this type of help %c\n", type);
  }
  }
}



void do_option(va_alist)
     va_dcl
{
  va_list args;
  int code, action;

  va_start(args);
  code = va_arg(args, int);
  action = va_arg(args, int);
```

```
  (*cit[code].f)(OPTT, action, args);
  va_end(args);
}


void do_variable(va_alist)
     va_dcl
{
  int code;
  int action;
  va_list args;


  va_start(args);
  code = va_arg(args, int);
  action = va_arg(args, int);

  if ((!cit[code].empty) && ((cit[code].type & VART) == VART))
    (*cit[code].f)(VART, action, args);
}


void do_breakpoint(va_alist)
     va_dcl
{
  int code;
  int action;
  va_list args;


  code = va_arg(args, int);
  action = va_arg(args, int);

  if ((!cit[code].empty) && ((cit[code].type & BREAKT) == BREAKT))
    (*cit[code].f)(BREAKT, action, args);
}


void do_user(va_alist)
     va_dcl
{
  int code;
  int action;
  va_list args;


  code = va_arg(args, int);
  action = va_arg(args, int);

  if (!cit[code].empty)
    (*cit[code].f)(USERT, action, args);
}



void sig_hdl()
{
  ckpt_periodic_request();
```

```
  signal(SIGTERM, sig_hdl);
  signal(SIGUSR1, sig_hdl);
}


void sig_alrm()
{
  static char a[3][3] = { "opt", "CA", "0" };

  c_opt(CMNDT, XCMND, 3, a);
  ckpt_periodic_request();
  signal(SIGALRM, SIG_DFL);
}

/**********************************************************************/


void c_file(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  register int i, j;
  TABLE tbl;
  char *cmfn, *sfn, f1[256], f2[256];
  FILESPEC *fs;


  switch (action) {

  case XCMND: {
    switch (ac) {

    case 1: { /* no args -> list al open files */

      if (num_files == 0)
           printf("no open statistice files\n");

      else {
           printf("open statistics files:\n");
           for(i = 0; i < CITSZ; i++)
             if ((!cit[i].empty) && ((cit[i].type & FILET) == FILET))
               printf("%s\n", cit[i].name);
      }
      break;
    }

    case 2: { /* one arg -> list specified open file */

      printf("\n");
      if ((j = find_in_cit(av[i], FILET)) == -1)
           printf("file:  file %s not open\n", av[i]);
      else
           printf("%s\n", cit[j].name);
      break;
    }

    case 4:
    case 5: { /* 4/5 args -> open specified statistics file */
```

```
    if ((!strcmp(av[2], "NULL")) && (!strcmp(av[3], "NULL"))) {
        printf("file:  must have serial and/or cm\n");
        return;
    }

    if ((!strcmp(av[3], "NULL")) && !shapeset) {
        printf("file:  must set lattice size first\n");
        return;
    }

    if ((tbl.name = strdup(av[1])) == NULL) {
        fprintf(stderr, "\n\n\n\nstrdup failed\n\n\n\n");
        exit(0);
    }

    tbl.type = FILET;
    tbl.data = Malloc(sizeof(FILESPEC));
    fs = (FILESPEC *) tbl.data;
    tbl.f = NULL;

    fs->flint = ((ac == 5) ? atoi(av[4]) : 0);

    sprintf(f1, "%s/statistics/%s", DVHOME, av[3]);
    sprintf(f2, "%s/statistics/%s", FEHOME, av[2]);

    cmfn = ((!strcmp(av[3], "NULL")) ? NULL : f1);
    sfn = ((!strcmp(av[2], "NULL")) ? NULL : f2);

    if ((fs->fd = bopen(cmfn, sfn, "w", fs->flint)) == NULL) {
        printf("file:  couldn't open file (%s, %s)\n", cmfn, sfn);
        free(tbl.data);
        free(tbl.name);
        return;
    }

    if(fs->fd->cm < 0)
        strcpy(f1, "NONE");

    if(!(fs->fd->s))
        strcpy(f2, "NONE");

    reg_one_spec(tbl, NULL);
    printf("file:  opened file (%s, %s) as %s with flush interval of %d\n",
           f1, f2, av[1], fs->flint);

    num_files++;
    break;
  }

  default: { /* any other number of args is an error */
    printf("file:  error, see help for file command\n");
  }
  }
  break;
}

case HELP: {
  print_help(FILE_HELP, HELPDIR, "file");
```

```
    break;
  }
  }
}

void c_cfile(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  register int i, j, k;
  FILESPEC *fs;
  VARSPEC vs;

  switch(action) {

  case XCMND: {
    if (check_args("cfile", 0, 2, ac)) {

      printf("cfile:\n");
      for(i = 1; i < ac; i++) {

            if ((j = find_in_cit(av[i], FILET)) == -1)
              printf("lfile:  file %s not open\n", av[i]);
            else {
              fs = (FILESPEC *) cit[j].data;
              bclose(fs->fd);

              /* Do DACTVAR for eact var associated */
              for(k = 0; k < CITSZ; k++)
                if ((!cit[k].empty) && ((cit[k].type & VART) == VART)) {
                  do_variable(k, GETVAR, &vs);
                  if (j == vs.fsc)
                        do_variable(k, DACTVAR);
                }
              free(cit[j].name);
              free(cit[j].data);
              cit[j].empty = -1;
              printf("cfile:  file %s closed\n", cit[j].name);
              num_files--;
            }
      }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "cfile");
    break;
  }
  }
}
```

```
void c_assoc(type, action, ac, av)
    int type, action, ac;
    char **av;
{
  int vc, fc;
  VARSPEC vs;
  register int i, j;


  switch(action) {

  case XCMND: {
    switch (ac) {

    case 1: { /* no args -> list all associations */

      printf("Current Associations:\n");

      for(i = 0; i < CITSZ; i++) {
            if ((!cit[i].empty) && ((cit[i].type & FILET) == FILET)) {

              printf("%s:  ", cit[i].name);

              for(j = 0; j < CITSZ; j++)
                if ((!cit[j].empty) && ((cit[j].type & VART) == VART)) {
                  do_variable(j, GETVAR, &vs);
                  if (i == vs.fsc)
                        printf("%s  ", cit[j].name);
                }
              printf("\n");
            }
      }
      break;
    }

    default: { /* create an association */

      if (check_args("assoc", 1, 3, ac)) {

            if ((vc = find_in_cit(av[1], VART)) == -1) {
              printf("assoc:  no such variable %s\n", av[1]);
              return;
            }

            if ((fc = find_in_cit(av[2], FILET)) == -1) {
              printf("assoc:  no such statistics file %s\n", av[2]);
              return;
            }

            do_variable(vc, GETVAR, &vs);
            if (vs.fsc >= 0) {
              printf("assoc:  variable already associated with a file\n");
              return;
            }

            vs.fsc = fc;
            vs.set = 0;

            do_variable(vc, SETVAR, &vs);
```

```
        printf("assoc:  variable %s associated with file %s\n",
            cit[vc].name, cit[fc].name);
      }
    }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "assoc");
    break;
  }
  }
}




void c_dassoc(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  int vc;
  VARSPEC vs;

  switch(action) {

  case XCMND: {
    if (check_args("dassoc", 1, 2, ac)) {

      if ((vc = find_in_cit(av[1], VART)) == -1) {
        printf("dassoc:  no such variable %s\n", av[1]);
        return;
      }

      do_variable(vc, GETVAR, &vs);

      if (vs.fsc < 0)
        printf("dassoc:  variable not associated with any file\n");
      else {
        vs.fsc = -1;
        vs.set = 0;
        do_variable(vc, SETVAR, &vs);
      }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "dassoc");
    break;
  }
  }
}
```

```
void c_var(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  register int vc, i;
  VARSPEC vs;


  switch(action) {

  case XCMND: {
    switch (ac) {

    case 1: { /* no args -> list all variables */

      if (num_vars) {

            printf("Statistics variables available:\n");

            for(i = 0; i < CITSZ; i++)
              if ((!cit[i].empty) && ((cit[i].type & VART) == VART))
                printf("%s\n", cit[i].name);

            printf("\n");
      }

      else
            printf("No statistics variables available\n");

      break;
    }

    default: { /* activate/deactivate a variable */

      if (check_args("var", 1, 3, ac)) {

            if ((vc = find_in_cit(av[1], VART)) == -1) {
              printf("var:  no such variable %s\n", av[1]);
              return;
            }

            do_variable(vc, GETVAR, &vs);

            if (vs.fsc < 0) {
              printf("variable is not associated with a file yet\n");
              return;
            }

            if (!strcmp(av[2], "on")) {
              vs.set = -1;
              do_variable(vc, SETVAR, &vs);
              do_variable(vc, ACTVAR);
              printf("var:  variable %s activated\n", av[1]);
              return;
            }

            if (!strcmp(av[2], "off")) {
              vs.set = 0;
```

```
            do_variable(vc, SETVAR, &vs);
            do_variable(vc, DACTVAR);
            printf("variable %s deactivated\n", av[1]);
            return;
          }

          printf("var:  specify \"on\" or \"off\"\n");
      }
    }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "var");
    break;
  }
  }
}



void c_source(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    if (check_args("source", 1, 2, ac))
      parse_file(av[1]);
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "source");
    break;
  }
  }
}



void c_check(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    check_args("check", 0, 0, ac);
    if (ckpt(feprefix, cmprefix) < 0)
      printf("Checkpoint failed because:\n%s", ckpt_errormsg);
    break;
  }
```

```
  case HELP: {
    print_help(FILE_HELP, HELPDIR, "check");
    break;
  }
  }
}



void c_break(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  register int bc, i;
  BREAKSPEC bs;

  switch(action) {

  case XCMND: {
    switch (ac) {

    case 1: { /* no args -> list all breakpoints */

      if (num_breaks) {
            printf("Breakpoints available:\n");

            for(i = 0; i < CITSZ; i++)
              if ((!cit[i].empty) && ((cit[i].type & BREAKT) == BREAKT))
                printf("%s\n", cit[i].name);
      }

      else
            printf("no breakpoints available\n");

      break;
    }


    default: { /* activate/deactivate breakpoints */

      if (check_args("break", 1, 3, ac)) {

            if ((bc = find_in_cit(av[1], BREAKT)) == -1) {
              printf("break:  no such breakpoint %s\n", av[1]);
              return;
            }

            do_breakpoint(bc, GETBRK, &bs);

            if (!strcmp(av[2], "on")) {
              bs.set = -1;
              do_breakpoint(bc, SETBRK, &bs);
              printf("break:  breakpoint %s activated\n", av[1]);
              return;
            }

            if (!strcmp(av[2], "off")) {
              bs.set = 0;
```

```
          do_breakpoint(bc, SETBRK, &bs);
          printf("breakpoint %s deactivated\n", av[1]);
          return;
        }

        printf("break:  specify \"on\" or \"off\"\n");
    }
   }
   }

   break;
  }

  case HELP: {
   print_help(FILE_HELP, HELPDIR, "break");
   break;
  }
  }
}




void c_opt(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  int oc, i;


  switch(action) {

  case XCMND: {
   switch (ac) {

   case 1: { /* no args -> list all options that can be set */

     printf("Options available:\n\n");
     for(i = 0; i < CITSZ; i++)
        if((!cit[i].empty) && ((cit[i].type & OPTT) == OPTT)) {
          printf("%s:   ", cit[i].name);
          do_option(i, PRTOPT);
        }
     break;
   }

   case 2: { /* print value of an option */

     if ((oc = find_in_cit(av[1], OPTT)) == -1) {
        printf("opt:  no such option %s\n", av[1]);
        return;
     }

     do_option(oc, PRTOPT);
     break;
   }
```

```
    default: { /* set option value */
      OPTSPEC os;

      if (check_args("opt", 0, 3, ac)) {

            if ((oc = find_in_cit(av[1], OPTT)) == -1) {
              printf("opt:  no such option %s\n", av[1]);
              return;
            }

            do_option(oc, GETOPT, &os);
            if (!running || !os.vital)
              do_option(oc, SETOPT, ac - 2, av + 2);
            else if (running) {
              printf("You cannot change essential options once the simulation has begun.\n");
              printf("Issue a reset command to terminate the simulation first.\n");
            }
      }
    }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "opt");
    break;
  }
  }
}



void c_log(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    printf("log is not available yet\n");
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "log");
    break;
  }
  }
}



void c_echo(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {
```

```
  case XCMND: {
    switch (ac) {

    case 1: { /* print echo status */

      printf("script file echo is %s\n", (cmnd_echo) ? "on" : "off");
      break;
    }

    default: { /* set echo status */

      if (check_args("echo", 1, 2, ac)) {

          if (!strcmp(av[1], "on")) {
            printf("echo:  turning script file echo on\n");
            cmnd_echo = -1;
          }

          else if (!strcmp(av[1], "off")) {
            printf("echo:  turning script file echo off\n");
            cmnd_echo = 0;
          }

          else
            printf("echo:  specify one of on or off\n");
      }
    }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "echo");
    break;
  }
  }
}
```

```
void c_help(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  register int i;
  int cc;
  char cmd[256];
  void (*f)();


  switch(action) {

  case XCMND: {
    switch(ac) {

    case 1: { /* list names of all commands */
```

```
      for(i = 0; i < CITSZ; i++)
            if (!(cit[i].empty) && ((cit[i].type & CMNDT) == CMNDT))
              printf("%s\n", cit[i].name);
      break;
    }

    default: { /* one or more commands specified */

      for (i = 1; i < ac; i++) {

            if ((cc = find_in_cit(av[i], CMNDT)) == -1) {
              printf("\nhelp:  no such command %s\n", av[i]);
              continue;
            }

            (*cit[cc].f)(CMNDT, HELP);
      }
    }
    }
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "help");
    break;
  }
  }
}



void c_cont(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    check_args("cont", 0, 0, ac);

    if (running) {
      exit_cmnd_mode = -1;
      return;
    }
    else {
      printf("Use run command to start the simulation\n");
      return;
    }

    if (ignore_run) {
      printf("Cannot enter simulation mode:  cont command ignored\n");
      return;
    }

    if (!cont_safe()) {
      printf("cont:  It is not safe to cont, specify values for essential options\n");
      c_opt(CMNDT, XCMND, 1, NULL);
      exit_cmnd_mode = 0;
```

```
      return;
    }

    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "run");
    break;
  }
  }
}




void c_run(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    check_args("run", 0, 0, ac);

    if (!running && ignore_run) {
      printf("Cannot enter simulation mode:  run command ignored\n");
      return;
    }

    if (!cont_safe()) {
      printf("cont:  It is not safe to run, specify values for essential options\n");
      c_opt(CMNDT, XCMND, 1, NULL);
      exit_cmnd_mode = 0;
      return;
    }

    if (running) {
      exit_cmnd_mode = -1;
      return;
    }

    running = -1;

    /* same as with(lattice) */
    CM_set_vp_set(lattice);

    /* Initialize parallel memory alocation package */
    PMemInit();

    /* Init bishop, the model and and utility code */
    printf("CM memory:  %d bits\n\n", CM_available_memory());
    printf("Initializing Bishop\n");
    init_bishop(ac_save, av_save, FINAL);
    printf("CM memory:  %d bits\n\n", CM_available_memory());
    printf("Initializing model\n");
    init_model(ac_save, av_save, FINAL);
    printf("CM memory:  %d bits\n\n", CM_available_memory());
```

```
    printf("Initializing utilities\n");
    init_util(ac_save, av_save, FINAL);
    printf("CM memory:  %d bits\n\n", CM_available_memory());

    run_bishop(ac_save, av_save);

    running = 0;
    break;
  }

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "run");
    break;
  }
  }
}



void c_quit(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND:
    death(0);

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "quit");
    break;
  }
  }
}



void c_reset(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND: {
    check_args("reset", 0, 0, ac);

    finish_util();
    finish_model();
    finish_bishop();
    finish_command();
    init_command(ac_save, av_save);
    init_bishop(ac_save, av_save, PRELIM);
    init_model(ac_save, av_save, PRELIM);
    init_util(ac_save, av_save, PRELIM);
    startup(1);
    longjmp(bishop_env, 1);
  }

  case HELP: {
```

```
    print_help(FILE_HELP, HELPDIR, "reset");
    break;
  }
  }
}


void c_load(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND:
    printf("load is not available yet\n");

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "load");
    break;
  }
  }
}


void c_save(type, action, ac, av)
     int type, action, ac;
     char **av;
{
  switch(action) {

  case XCMND:
    printf("save is not available yet\n");

  case HELP: {
    print_help(FILE_HELP, HELPDIR, "save");
    break;
  }
  }
}
```

B.7 bstdio.h

```
#ifndef BSTDIO_H_INCLUDED
#define BSTDIO_H_INCLUDED

#include <stdio.h>
#include <cm/cm_file.h>


typedef struct b_file {
  int cm;
  FILE *c, *s;
```

```
  int cnt, fi;
} BFILE;


BFILE *bopen();
void bclose();
void bflush();
int bprintf();
int bscanf();

#endif
```

B.8 bstdio.c

```
#include <stdio.h>
#include <varargs.h>
#include <cm/paris.h>
#include <cm/cm_errno.h>
#include <errno.h>

#include <mem.h>
#include <default/bstdio.h>


/* fprintf/fscanf like functions to write/read to serial and dv files
   includes a control file
   Format Specifiers:

   %K − write keyword to control file
   %D − write 32 bit integer to serial file
   %C − write 8 bit char to serial file
   %S − write string to serial file
   %G − write 64 bit integer to serial file
   %F − write 32 bit float to serial file
   %d − write 32 bit integer to dv file
   %c − write 8 bit char to dv file
   %s − write string to dv file
   %g − write 64 bit integer to dv file
   %f − write 32 bit float to dv file

   The specifiers for the dv file may be followed by a size modifer.
   Size modifers may be given fully or partially.  Partial modifiers
   casue the size to be taken from the parameter list directly before
   the data to be writen/read.

   [] − partial modifier, ex: %d[]
   [G] − full modifier indicating G copies of the modified specifier, ex:
          %d[2]

   These accomplish the same thing:

     bprintf(file, "%K%d[]", "TEST", 2, 5, 6);
     bprintf(file, "%K%d[2]", "TEST", 5, 6);
```

```
*/

BFILE *bopen(cmfn, sfn, mode, fi)
     char *cmfn, *sfn;
     char *mode;
     int fi;
{
  BFILE *file;
  int cmoflgs = 0;
  int cmmode = 0000666;
  char serial[100], control[100];



  file = (BFILE *) Malloc(sizeof(BFILE));
  file->cm = file->c = file->s = NULL;

  sprintf(serial, "%s.SERIAL", sfn);
  sprintf(control, "%s.CONTROL", sfn);

  if (!strcmp(mode, "r"))
    cmoflgs = CMFS_O_RDONLY;

  if (!strcmp(mode, "r+"))
    cmoflgs = CMFS_O_RDWR;

  if (!strcmp(mode, "w"))
    cmoflgs = CMFS_O_WRONLY | CMFS_O_TRUNC | CMFS_O_CREAT;

  if (!strcmp(mode, "w+"))
    cmoflgs = CMFS_O_RDWR | CMFS_O_TRUNC | CMFS_O_CREAT;

  if (!strcmp(mode, "a"))
    cmoflgs = CMFS_O_APPEND;

  if (!strcmp(mode, "a+"))
    cmoflgs = CMFS_O_APPEND | CMFS_O_TRUNC | CMFS_O_CREAT;


  file->fi = fi;
  file->cnt = 0;

  if ((file->c = fopen(control, mode)) == NULL) {
    free(file);
    return(NULL);
  }

  if (cmfn)
    if ((file->cm = CMFS_open(cmfn, cmoflgs, cmmode)) == -1)
      CMFS_perror("bopen");

  if (sfn)
    if ((file->s = fopen(serial, mode)) == NULL)
      perror("bopen");

  if (!(file->cm) && !(file->s))
      return(NULL);
  else
```

```
    return(file);
}


void bclose(file)
     BFILE *file;
{
  if (file) {

    if (file->cm)
      CMFS_close(file->cm);
    if (file->c)
      fclose(file->c);
    if (file->s)
      fclose(file->s);

    free(file);
  }
}



int bprintf(file, format, va_alist)
     BFILE *file;
     char *format;
     va_dcl
{
  register int fh = 0, fl;
  va_list args;
  int s;
  int sitm = 0, pitm = 0, b;
  char *cp, c, o;
  int *ip, i;
  float *fp, f;
  double *dp, d;
  CM_field_id_t cmfid;


  va_start(args);

  if (!format) {
    fprintf(stderr, "bprintf:  NULL format string\n");
    return(-1);
  }

  if ((fl = strlen(format)) < 1) {
    fprintf(stderr, "bprintf:  zero length format string\n");
    return(-1);
  }

  if(!file) {
    fprintf(stderr, "bprintf:  NULL bfile pointer\n");
    return(-1);
  }

  if(!file->c) {
    fprintf(stderr, "bprintf:  NULL control file descriptor\n");
    return(-1);
```

```
}

do {

  switch(format[fh++]) {

  case '%': { /* begining of format specifier */

    /* figure out which format we are dealing with */

    switch(format[fh++]) {

    case 'K': { /* Keyword */

          cp = va_arg(args, char *);
          if (fprintf(file->c, "%s ", cp) == EOF)
            goto RET_ERROR;

          break;
    }

    case 'S': { /* String (serial) */

          cp = va_arg(args, char *);

          if (fprintf(file->c, "%%S ") != 3)
            goto RET_ERROR;

          sitm += (b = fprintf(file->s, "%s", cp));
          if (b == EOF)
            goto RET_ERROR;

          break;
    }

    case 'D': { /* 32 bit signed integer (serial) */

          i = va_arg(args, int);

          if (fprintf(file->c, "%%D ") != 3)
            goto RET_ERROR;

          sitm += (b = fprintf(file->s, "%d", i));
          if (b == EOF)
            goto RET_ERROR;

          break;
    }

    case 'F': { /* 32 bit float (serial) */

          f = va_arg(args, float);

          if (fprintf(file->c, "%%F ") != 3)
            goto RET_ERROR;

          sitm += (b = fprintf(file->s, "%f", f));
          if (b == EOF)
            goto RET_ERROR;
```

```
    break;
}

case 'G': { /* 64 bit double (serial) */

    d = va_arg(args, double);

    if (fprintf(file->c, "%%G ") != 3)
      goto RET_ERROR;

    sitm += (b = fprintf(file->s, "%lf", d));
    if (b == EOF)
      goto RET_ERROR;

    break;
}

case 'C': { /* 8 bit character */

    c = va_arg(args, char);

    if (fprintf(file->c, "%%C ") != 3)
      goto RET_ERROR;

    sitm += (b = fprintf(file->s, "%c", c));
    if (b == EOF)
      goto RET_ERROR;

    break;
}

case 's': /* string (parallel) */
case 'd': /* 32 bit integer (parallel) */
case 'f': /* 32 bit float (parallel) */
case 'g': /* 64 bit double (parallel) */
case 'c': /* 8 bit character (parallel) */
case 'b': /* 1 bit boolean (parallel only) */
    {

      o = format[fh-1];
      s = 1;

      if ((o == 's') && ((fh >= fl) || (format[fh] != '['))) {
        fprintf(stderr, "bprintf:  format string error %s\n", format);
        goto RET_ERROR;
      }

      else if (format[fh] == '[') {

        if (format[++fh] == ']') { /* size is provied as arg */

          s = va_arg(args, int);
          fh++;
        }

        else { /* size provided in format string */

          if (sscanf(format+fh, "%d]", &s) != 1) {
```

```
          fprintf(stderr, "bprintf:  format string error %s\n", format);
          goto RET_ERROR;
    }

    do { fh++; } while (format[fh] != ']');
    fh++;
  }
}

cmfid = va_arg(args, CM_field_id_t);

switch(o) {

case 's':
case 'c': {

  if (fprintf(file->c, "%%%c[%d] ", o, s) == EOF)
    goto RET_ERROR;

  pitm += (b = CMFS_write_file_always(file->cm, cmfid, s * 8));
  if (b != s * 8)
    goto RET_ERROR;

  break;
}

case 'd':
case 'f': {

  if (fprintf(file->c, "%%%c[%d] ", o, s) == EOF)
    goto RET_ERROR;

  pitm += (b = CMFS_write_file_always(file->cm, cmfid, s * 32));
  if (b != s * 32)
    goto RET_ERROR;

  break;
}

case 'g': {

  if (fprintf(file->c, "%%g[%d] ", s) == EOF)
    goto RET_ERROR;

  pitm += (b = CMFS_write_file_always(file->cm, cmfid, s * 64));
  if (b != s * 64)
    goto RET_ERROR;

  break;
}

case 'b': {

  if (fprintf(file->c, "%%b[%d] ", s) == EOF)
    goto RET_ERROR;

  pitm += (b = CMFS_write_file_always(file->cm, cmfid, s));
  if (b != s)
    goto RET_ERROR;
```

```
          break;
        }
        }

        break;
      }

  case '%':{ /* literal % */

      sitm += (b = fprintf(file->s, "%%"));
      if (b == EOF)
        goto RET_ERROR;

      break;
  }

  default: {
      fprintf(stderr, "bprintf:  format string error %s\n", format);
      goto RET_ERROR;
      break;
  }
  }

  break;
 }

 default: {
  /* Write all other characters to the SERIAL file */

  sitm += (b = fprintf(file->s, "%c", format[fh-1]));
  if (b == EOF)
      goto RET_ERROR;
 }
 }

} while (fh < fl);

fprintf(file->c, "(%d, %d)\n", sitm, pitm);

va_end(args);
if (file->fi)
  if(! ((file->cnt++) % file->fi))
    bflush(file);
return(0);

RET_ERROR:
fprintf(file->c, "(%d, %d)\n", sitm, pitm);

if (CMFS_errno)
  CMFS_perror("bprintf");
perror("bprintf");

va_end(args);

return(-1);
}
```

```
int bscanf(file, format, sitm, pitm, va_alist)
     BFILE *file;
     int *sitm, *pitm;
     char *format;
     va_dcl
{
  register int fh = 0;
  int fl;
  va_list args;
  int s, b;
  char *cp, c, o;
  int *ip, i;
  float *fp, f;
  double *dp, d;
  CM_field_id_t cmfid;


  *sitm = *pitm = 0;

  va_start(args);

  if ((fl = strlen(format)) < 1)
    return(0);

  do {

    switch(format[fh++]) {

    case '%': { /* begining of format specifier */

      /* figure out which format we are dealing with */

      switch(format[fh++]) {

      case 'S': { /* String (serial) */

            cp = va_arg(args, char *);

            *sitm += (b = fscanf(file->s, "%s", cp));
            if (b != 1)
              goto RET_ERROR;

            break;
      }

      case 'D': { /* 32 bit signed integer (serial) */

            ip = va_arg(args, int *);

            *sitm += (b = fscanf(file->s, "%d", ip));
            if (b != 1)
              goto RET_ERROR;

            break;
      }
```

```
case 'F': { /* 32 bit float (serial) */

    fp = va_arg(args, float *);

    *sitm += (b = fscanf(file->s, "%f", fp));
    if (b != 1)
      goto RET_ERROR;

    break;
}

case 'G': { /* 64 bit double (serial) */

    dp = va_arg(args, double *);

    *sitm += (b = fscanf(file->s, "%lf", dp));
    if (b != 1)
      goto RET_ERROR;

    break;
}

case 'C': { /* 8 bit character */

    cp = va_arg(args, char *);

    *sitm += (b = fscanf(file->s, "%c", cp));
    if (b != 1)
      goto RET_ERROR;

    break;
}

case 's': /* string (parallel) */
case 'd': /* 32 bit integer (parallel) */
case 'f': /* 32 bit float (parallel) */
case 'g': /* 64 bit double (parallel) */
case 'c': /* 8 bit character (parallel) */
case 'b': /* 1 bit boolean (parallel only) */
    {

      o = format[fh-1];
      s = 1;

      if ((o == 's') && ((fh >= fl) || (format[fh] != '['))) {
        fprintf(stderr, "bscanf:  format string error %s\n", format);
        goto RET_ERROR;
      }

      else if (format[fh] == '[') {

        if (format[++fh] == ']') { /* size is provied as arg */

          s = va_arg(args, int);
          fh++;
        }

        else { /* size provided in format string */
```

```
      if (sscanf(format+fh, "%d]", &s) != 1) {
        fprintf(stderr, "bscanf:  format string error %s\n", format);
        goto RET_ERROR;
      }

      do { fh++; } while (format[fh] != ']');
      fh++;
    }
   }

   cmfid = va_arg(args, CM_field_id_t);

   switch(o) {

   case 's':
   case 'c': {

     *pitm += (b = CMFS_read_file_always(file->cm, cmfid, s * 8));
     if (b != s * 8)
       goto RET_ERROR;

     break;
   }

   case 'd':
   case 'f': {

     *pitm += (b = CMFS_read_file_always(file->cm, cmfid, s * 32));
     if (b != s * 32)
       goto RET_ERROR;

     break;
   }

   case 'g': {

     *pitm += (b = CMFS_read_file_always(file->cm, cmfid, s * 64));
     if (b != s * 64)
       goto RET_ERROR;

     break;
   }

   case 'b': {

     *pitm += (b = CMFS_read_file_always(file->cm, cmfid, s));
     if (b != s)
       goto RET_ERROR;

     break;
   }
   }

   break;
  }

default: {
  fprintf(stderr, "bscanf:  format string error %s\n", format);
  goto RET_ERROR;
```

```
          break;
      }
      }

      break;
    }


    default: { /* didn't match any format specifiers so its an error */
      fprintf(stderr, "bscanf:  format string error %s\n", format);
      goto RET_ERROR;
      break;
    }
    }

  } while (fh < fl);

  fscanf(file->c, "\n");

  va_end(args);
  return(0);

 RET_ERROR:
  if (CMFS_errno)
    CMFS_perror("bscanf");
  return(-1);
}



void bflush(file)
     BFILE *file;
{
  if (file->cm)
    CMFS_flush(file->cm);

  if (file->s)
    fflush(file->s);

  if (file->c)
    fflush(file->c);
}
```

B.9 argvize.c

```
#include <string.h>
#include <mem.h>



char **argvize(in, ac)
     char *in;
```

```
    int *ac;
{
  char **av;
  int i, len;

  *ac = 0;
  av = (char **) Calloc(40, sizeof(char *));

  av[*ac] = strtok(in, " \n");
  if (! av[*ac]) {
    free(av);
    return(NULL);
  }

  (*ac)++;

  while( (av[*ac] = strtok(NULL, " \n")) != NULL )
    (*ac)++;

  av = (char **) Realloc(av, *ac * sizeof(char *));

  return(av);
}
```

B.10 hdbmo.cs

```
#include <math.h>
#include <varargs.h>

#include <pmem.hs>
#include <default/gcoord.hs>
#include <default/rnd.hs>
#include <default/bstdio.h>

#include <default/command.h>
#include <bishop.hs>
#include <hdb/hdbmo.hs>

typedef struct hist1 {
  float low;
  float high;
} HIST;

/* Price history of lowest and highest prices seen and experienced */
static HIST:lattice *seen, *past;

static int i1 = 0, i2 = 0;

static TABLE hdb_extensions[] = {
  { "DESIRE", 0, VART, hdl_desire, NULL },
  { NULL, NULL, NULL, NULL, NULL }
};

int hdbexts[10];
```

```
int:lattice *desire = (int:lattice *) NULL;

void increase_10per(int), decrease_10per(int);




model_intro()
{
  printf("MODEL:\tHedonistic Dain Bramaged\n");
  printf("\tMarket Omnipotence\n");
}




void choose_merchant()
{
  register int j;
  int who, xi, yi, zi;
  float lowp;

  /* choose agent with the lowest price <= my cutoff */
  for (j = 0; j < G; j++) {

    plan[j].flag = HIGH_PRICES;
    plan[j].who = NO_ONE;

    where((plan[j].disp == SELL) & (plan[j].price == seen[j].low))
      who = (int) SELF_ID;

    where((plan[j].disp == BUY) & (plan[j].price >= seen[j].low)) {
      plan[j].flag = NADA;
      plan[j].who = who;
      plan[j].price = seen[j].low;
      plan[j].quant = desire[j];
    }

  } /* for (j = 0; */
}



void uds()
{
  register int j;
  float lowp, highp;


  /* update histogram of highest and lowest prices recieved */

  for (j = 0; j < G; j++) {

    where(plan[j].disp == SELL) {
      lowp = <?= plan[j].price;
      highp = >?= plan[j].price;
    }

    seen[j].low = (lowp < seen[j].low ? lowp : seen[j].low);
```

```
    seen[j].high = (highp > seen[j].high ? highp : seen[j].high);
  }
}



void uss(int:lattice x, int:lattice y, int:lattice z, int j)
{
  where([x][y][z]past[j].low > act[j].price) {
    [x][y][z]past[j].low = act[j].price;
  }

  where ([x][y][z]past[j].high < act[j].price) {
    [x][y][z]past[j].high = act[j].price;
  }

  where(past[j].low > act[j].price) {
    past[j].low = act[j].price;
  }

  where (past[j].high < act[j].price) {
    past[j].high = act[j].price;
  }
}






void flip_flop_null()
{
  /* Do absolutely NOTHING */
}





void flip_flop()
{
  register int j;
  bool:lattice a;


  for (j = 0; j < G; j++) {
    a = 1;

    where ((a == 1) & (plan[j].disp == BUY)) {

      where (act[j].disp == FAIL) {
            /* couldn't buy any so sell next period */
            plan[j].disp = SELL;
            plan[j].price = (seen[j].high != 0.0F ? seen[j].high : plan[j].price);
            plan[j].quant = (desire[j] < inv[j]) ? desire[j] : inv[j];
            a = 0;
      }
    }

    where ((a == 1) & (plan[j].disp == SELL)) {
```

```
      where (act[j].disp == FAIL) {
            decrease_10per(j);
            a = 0;
      }

      where ((act[j].disp == SOLD) & (inv[j] != 0)) {
            increase_10per(j);
            a = 0;
      }

      where ((act[j].disp == SOLD) & (inv[j] == 0)) {
            plan[j].disp = BUY;
            plan[j].price = (past[j].low + past[j].high * 1.1F) / 2.0F;
            plan[j].quant = desire[j];
            a = 0;
      }
    }

    where ((a == 1) & (plan[j].disp == HOLD)) {
      plan[j].disp = BUY;
      a = 0;
    }

  }
}



void decrease_10per(int g)
{
  plan[g].price = plan[g].price * 0.9;
  plan[g].quant = (desire[g] < inv[g]) ? desire[g] : inv[g];

  where (plan[g].price < 10.0F) {
    plan[g].disp = BUY;
    plan[g].price = (past[g].high != 0.0F ?
                          (past[g].low + past[g].high * 1.1F) / 2.0 : 50.0F);
    plan[g].quant = desire[g];
  }
}


void increase_10per(int g)
{
  plan[g].price = plan[g].price * 1.1F;
  plan[g].quant = (desire[g] < inv[g]) ? desire[g] : inv[g];
}



void init_model(int ac, char *av[], int step)
{
  register int h;


  switch (step) {

  case PRELIM: {
```

```
    if (!i1) {

      /* Parse command line options */
      if (ac != 1) {
            for (h=1; h<ac; h++) {
            }
      }

      /* register options */
      reg_spec(hdb_extensions, hdbexts);

      update_diff_stats = uds;
      update_sale_stats = uss;
      calculate_expectations = flip_flop;

      i1 = -1;
    }
    break;
  }

  case FINAL: {

    if (!i2) {

      desire = (int:lattice *) PMalloc(G * boolsizeof(int:lattice));
      past = (HIST:lattice *) PMalloc(G * boolsizeof(HIST:lattice));
      seen = (HIST:lattice *) PMalloc(G * boolsizeof(HIST:lattice));

      for (h = 0; h < G; h++) {
            desire[h] = frnd() * inv[h] + 1;
            past[h].low = seen[h].low = HUGE_VAL;
            past[h].high = seen[h].high = 0.0F;
            plan[h].price = 10.0 + frnd(101.0F) + frnd(60.0F);
            plan[h].quant = desire[h];

            where (rnd(512) > 256)
              plan[h].disp = SELL;

            else
              plan[h].disp = BUY;
      }

      i2 = -1;

      /* Write model parameters which never change */
      do_variable(EDESIRE, STATVAR);
    }
    break;
  }
  }
}


void finish_model()
{
  if (i2) {
```

```
    PFree(desire);
    PFree(past);
    PFree(seen);
  }

  i2 = 0;
}




/***********************************************************************/



void hdl_desire(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static char fn[256];
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "DESIRE");
      if ((ofd = fopen(fn, "w")) == NULL)
          printf("\n\nhdl_desire:  Couldn't open ouput file %s\n\n", fn);
    }
    else {
      if ((ofd = fopen(fn, "a+")) == NULL)
          fprintf(stderr, "\n\nhdl_desire:  Couldn't open ouput file %s\n\n", fn);
    }
    break;
  }

  case DACTVAR: {
    if (ofd)
      fclose(ofd);
    break;
  }

  case STATVAR: {
    register int j;
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *)cit[fsc].data;

      if (bprintf(fs->fd, "%K%d[]", "DESIRE", G, desire))
          printf("write of desire var failed\n");

      for(j=0; j<G; j++)
          fprintf(ofd, "%d %10.10f %d ",
                  (<?= desire[j]),
                  (+= desire[j]) / (float) (LS*LS*LS),
                  (>?= desire[j]));
```

```
    fprintf(ofd, "\n");
  }
  break;
}

case GETVAR: {
  VARSPEC *vs;

  next_arg(vs, args, VARSPEC *);

  vs->fsc = fsc;
  vs->set = set;
  break;
}

case SETVAR: {
  VARSPEC *vs;

  next_arg(vs, args, VARSPEC *);
  if (vs->fsc != NA)
    fsc = vs->fsc;

  if (vs->set != NA)
    set = vs->set;
  break;
}
}
}
```

B.11 musd.cs

```
#include <math.h>
#include <varargs.h>

#include <pmem.hs>
#include <default/gcoord.hs>
#include <default/rnd.hs>
#include <default/bstdio.h>

#include <default/command.h>
#include <bishop.hs>
#include <mu/musd.hs>
#include <default/pbsort.hs>


static int i1 = 0, i2 = 0;

static TABLE mu_extensions[] = {
  { "MU", 0, VART, hdl_MU, NULL },
  { "ETA", 0, VART, hdl_eta, NULL },
  { "EP", 0, VART, hdl_EP, NULL},
  { "OP", 0, VART, hdl_OP, NULL },
  { "LAMBDA", 0, VART, hdl_lambda, NULL },
  { "CS", 0, VART|OPTT, hdl_CS, NULL },
```

```
  { "DRATE", 0, VART|OPTT, hdl_drate, NULL },
  { NULL, NULL, NULL, NULL, NULL }
};

int muexts[10];


int CS;
int drate;

int:lattice *crib_count = (int:lattice *) NULL;
int:lattice *crib_count_cap = (int:lattice *) NULL;
int:lattice *crib_id = (int:lattice *) NULL;
bool:lattice *crib_g = (bool:lattice *) NULL;
float:lattice *MU = (float:lattice *) NULL;
float:lattice *eta = (float:lattice *) NULL;
float:lattice *EP = (float:lattice *) NULL;
float:lattice *OP = (float:lattice *) NULL;
float:lattice *lambda = (float:lattice *) NULL;



model_intro()
{
  printf("MODEL:\tMarginal Utility - Adaptive Price/Cutoff\n");
  printf("\tAdaptive E[p] - Slow Diffusion\n");
}



void choose_merchant()
{
  register int j, k;
  int:lattice xi, yi, zi;
  float:lattice lowp;

  /* choose agent with the lowest price <= my cutoff */
  for (j = 0; j < G; j++) {

    plan[j].who = NO_ONE;
    plan[j].flag = NO_ADS;

    where ((plan[j].disp == BUY) & (*crib_count != 0)) {

      lowp = plan[j].price;
      for (k = 0; k < CS; k++) {

            where (crib_g[k*CS+j] && (crib_id[k] >= 0)) {

              g_coord(crib_id[k], &xi, &yi, &zi, LS);

              where ([xi][yi][zi]plan[j].price <= lowp) {
                plan[j].who = crib_id[k];
                plan[j].flag = NADA;
                plan[j].quant = ((plan[j].quant <= [xi][yi][zi]plan[j].quant) ?
                                 plan[j].quant : [xi][yi][zi]plan[j].quant);
                lowp = [xi][yi][zi]plan[j].price;
              }
```

```
      } /* where (crib_g[k*CS+j] */
    } /* for (k = 0; */
  } /* where (plan[j].disp */

  where((plan[j].who == NO_ONE) & (*crib_count != 0))
    plan[j].flag = HIGH_PRICES;
 } /* for (j = 0; */
}



int:lattice comp_mu(int:lattice *r1, int:lattice *r2)
{
 int:lattice a = 0;

 where(MU[*r1] < MU[*r2])
   a = -1;

 where(MU[*r1] > MU[*r2])
   a = 1;

 return(a);
}



void construct_QR(int:lattice *r, int:lattice *sell)
{
 register int k;
 int:lattice hs, lb;

 /* *sell marks the boundry between goods being sold and bought */
 /* goods ranked from position 0 to *sell - 1 are being sold */
 /* goods ranked from position *sell to G - 1 are being bought */

 /* *r is an array of integers that represent the ranking of goods by MU */
 /* r[i] holds the index for the good ranked at the ith position */


 for (k = 0; k < G; k++) {

   /* Calcualte Q vector for this partition (buying) */
   hs = (*sell > 0) ? *sell - 1 : 0;
   where (k >= *sell) {
     plan[r[k]].disp = BUY;

     plan[r[k]].quant =
           abs( (int:lattice) ((eta[r[k]] * EP[r[hs]]) / (EP[r[k]] * eta[r[hs]]) *
                              inv[r[hs]] - inv[r[k]]) );

     /* if quant == 0 then we a we will just hold on to this good */
     where (plan[r[k]].quant == 0)
           plan[r[k]].disp = HOLD;
   } /* where (k >= *sell) */

   /* Calculate R vector for this partition (selling) */
   where (k < *sell) {
     lb = ((*sell < G) ? *sell : G - 1);
```

```
      plan[r[k]].disp = SELL;

      plan[r[k]].quant =
          abs( (int:lattice) (inv[r[k]] - (EP[r[lb]] * eta[r[k]]) /
                                (EP[r[k]] * eta[r[lb]]) * inv[r[lb]]) );

      /* if quant == 0 then we will just hold on to this good */
      where (plan[r[k]].quant == 0)
          plan[r[k]].disp = HOLD;

      /* sanity check, can't sell more than I have, don't want to sell all */
      where(plan[r[k]].quant >= inv[r[k]])
          plan[r[k]].disp = inv[r[k]] - 1;
    } /* where (k < *sell) */

    ckpt_periodic(feprefix, cmprefix);
  } /* for (k = 0; k < G; k++) */
}




void rank_mu()
{
  register int j, k;
  int oops;
  TRANS:lattice *oplan;
  int:lattice *r;
  bool:lattice found = 0;
  int:lattice selling = 0;
  float:lattice ROP, QEP;


  /* selling marks the boundry between goods being sold and bought */
  /* goods ranked from position 0 to selling - 1 are being sold */
  /* goods ranked from position selling to G - 1 and being bought */

  /* *r is an array of integers that represent the ranking of goods by MU */
  /* r[i] holds the index for the good ranked at the ith position */

  r = (int:lattice *) PMalloc(G * boolsizeof(int:lattice));
  oplan = (TRANS:lattice *) PMalloc(G * boolsizeof(TRANS:lattice));

  boolcpy((bool:lattice *) oplan, (bool:lattice *) plan,
            G * boolsizeof(TRANS:lattice));

  /* Calculate Marginal Utility per unit expected price for all goods */
  for (j = 0; j < G; j++) {
    MU[j] = eta[j] / (EP[j] * inv[j]);
    r[j] = j;
  }

  /* Write MU */
  do_variable(EMU, STATVAR);

  /* Sort from lowest to highest */
```

```
  pbsort((bool:lattice *) r, G, boolsizeof(int:lattice), comp_mu);

  /* Try partitions until we find one that works for every agent */
  /* The first one being to sell one good and buy all the others */
  do {

    /* Find Q and R vectors */
    where (!found) {
      construct_QR(r, &selling);

      /* Is this a feasible plan ? */
      ROP = QEP = 0.0F;
      for (k = 0; k < G; k++) {
            where(plan[k].disp == SELL)
              ROP += plan[k].quant * OP[k];
            where(plan[k].disp == BUY)
              QEP += plan[k].quant * EP[k];
      }

      /* Yes so we have found a partition that works */
      where((ROP + *(money)) >= QEP)
            found = 1;

      /* Otherwise try selling one more good and buying one less */
      else {
            selling++;

            where(selling > 20)
              if ((oops = CM_global_count_context()) != 0) {
                printf("\nrank_mu:  can't find partition for %d agents\n", oops);
                printf("dying in period %d\n", period);
                death(0);
              }
      }
    } /* where (!found) */

    ckpt_periodic(feprefix, cmprefix);

  } while ((&= found) == 0);

  for( k = 0; k< G; k++) {
    where((oplan[k].disp == SELL) & (plan[k].disp != SELL))
      plan[k].chng = 1;

    where(plan[k].disp == SELL)
      plan[k].price = OP[k];

    where(plan[k].disp == BUY)
      plan[k].price = EP[k];
  }

  PFree(r);
  PFree(oplan);
}
```

```
void ave_in_crib()
{
  register int k, j;
  int:lattice xi, yi, zi;




  for(k=0; k<CS; k++)
    for(j = 0; j < G; j++)
      where ((*crib_count < k) & (crib_g[k*CS+j])) {

            g_coord(crib_id[k], &xi, &yi, &zi, LS);

            EP[j] = (EP[j] + [xi][yi][zi]plan[j].price) / 2.0F;
      }
}



void adaptive()
{
  register int j, k;
  int:lattice xi, yi, zi;
  float:lattice lowp = (float:lattice) HUGE_VAL;


  /* Write current vlaues of EP and OP */
  do_variable(EEP, STATVAR);
  do_variable(EOP, STATVAR);

  /* for all goods */
  for (j = 0; j < G; j++) {

    /* We want to find the lowest price for this good on my cribsheet */
    /* We can use this to adjust our EP */

    /* where the crib sheet is not empty */
    where (*crib_count != 0) {

      /* for all crib entries */
      for(k = 0; k < CS; k++) {

            /* where the crib entry is not empty */
            where((crib_id[k] >= 0) & (crib_g[k*CS+j])) {

              /* get merchant's lattice address */
              g_coord(crib_id[k], &xi, &yi, &zi, LS);

              /* take lower of: lowp or merchant's price */
              lowp = ((lowp > [xi][yi][zi]plan[j].price) ?
                          [xi][yi][zi]plan[j].price :
                          lowp);
            } /* where(crib_id[k] >= 0) */
      } /* for(k = 0; k < CS; k++) */

      where ((plan[j].disp == BUY) & (act[j].disp == BOUGHT))
            lowp = ((lowp > act[j].price ? act[j].price : lowp));
```

```
      /* adjust EP by the lowest price seen */
      EP[j] += lambda[j] * (lowp − EP[j]);
    } /* where (*crib_count != 0) */

    /* adjust OP by our success in selling */
    where (plan[j].disp == SELL) {
      OP[j] += lambda[j] * OP[j] *
                ((act[j].quant − plan[j].quant) / plan[j].quant);

      /* If price has changed then mark it for cribsheet garbage collection */
      where(OP[j] != plan[j].price)
            plan[j].chng = 1;
    }

  } /* for (j = 0; j < G; j++) */

  /* Garbage collect prices that have changed */
  gc_cribs();
}




void init_model(int ac, char *av[], int step)
{
  register int h;


  switch (step) {

  case PRELIM: {

    if (!i1) {

      /* Parse command line options */
      if (ac != 1) {
            for (h=1; h<ac; h++) {
            }
      }

      /* register options */
      reg_spec(mu_extensions, muexts);

      rank_goods = rank_mu;
      calculate_expectations = adaptive;
      diffuse_info = slow_diffusion;

      i1 = −1;
    }
    break;
  }

  case FINAL: {

    if (!i2) {

      EP = (float:lattice *) PMalloc(G * boolsizeof(float:lattice));
```

```
      OP = (float:lattice *) PMalloc(G * boolsizeof(float:lattice));
      MU = (float:lattice *) PMalloc(G * boolsizeof(float:lattice));
      eta = (float:lattice *) PMalloc(G * boolsizeof(float:lattice));
      lambda = (float:lattice *) PMalloc(G * boolsizeof(float:lattice));

      crib_id = (int:lattice *) PMalloc(CS * boolsizeof(int:lattice));
      crib_g = (bool:lattice *) PMalloc(CS * G * boolsizeof(bool:lattice *));

      crib_count = (int:lattice *) PMalloc(boolsizeof(int:lattice));
      crib_count_cap = (int:lattice *) PMalloc(boolsizeof(int:lattice));

      /* Give agents a random distribution of expectatons and beliefs */
      /* Select eta and lambda randomly as well */
      for (h = 0; h < G; h++) {
            OP[h] = frnd(101.0F) + frnd(60.0F);
            EP[h] = frnd(101.0F) + frnd(60.0F);
            eta[h] = 0.001F + frnd();
            lambda[h] = 1.0F / frnd(100.0F);
      }

      *crib_count = 0;

      i2 = −1;

      /* Write model parameters which never change */
      do_variable(ECS, STATVAR);
      do_variable(EDRATE, STATVAR);
      do_variable(ELAMBDA, STATVAR);
      do_variable(EETA), STATVAR;
    }
    break;
  }
  }
}



void finish_model()
{
  if (i2) {
    CS = drate = 0;

    PFree(EP);
    PFree(OP);
    PFree(MU);
    PFree(eta);
    PFree(lambda);

    PFree(crib_id);
    PFree(crib_g);

    PFree(crib_count);
    PFree(crib_count_cap);
  }

  i2 = 0;
}
```

```
/*********************************************************************/

void hdl_CS(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;



  switch(type) {

  case OPTT: {
    switch (action) {

    case SETOPT: { /* write to option variable */
      int ac;
      char **av;

      next_arg(ac, args, int);
      next_arg(av, args, char **);

      if (ac == 1) {
            CS = atoi(av[0]);
            oset = -1;
      }

      break;
    }

    case PRTOPT: { /* print option variable */

      if (oset)
            printf("%d (cribsheet size) *\n", CS);
      else
            printf("unset (cribsheet size) *\n");

      break;
    }

    case GETOPT: {
      OPTSPEC *os;

      next_arg(os, args, OPTSPEC *);
      os->value = (char *) &CS;
      os->set = oset;
      os->vital = -1;
      break;
    }

    case OPTVITAL: {
      int *safe;

      next_arg(safe, args, int *);
      *safe &= oset;
      break;
    }
```

```
    }
    break;
  }

  case VART: {
    switch(action) {

    case STATVAR: {
      FILESPEC *fs;

      if (vset) {
            fs = (FILESPEC *) cit[fsc].data;

            if (bprintf(fs->fd, "%K%D\n", "CS", CS))
              printf("write of CS var failed");
      }
      break;
    }

    case GETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      vs->fsc = fsc;
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}




void hdl_drate(int type, int action, char *args)
{
  static int oset = 0, vset = 0;
  static int fsc = -1;



  switch(type) {

  case OPTT: {
```

```
  switch (action) {

  case SETOPT: { /* write to option variable */
    int ac;
    char **av;

    next_arg(ac, args, int);
    next_arg(av, args, char **);

    if (ac == 1) {
          drate = atoi(av[0]);
          oset = -1;
    }
    break;
  }

  case PRTOPT: { /* print option variable */

    if (oset)
          printf("%d (diffusion rate) *\n", drate);
    else
          printf("unset (diffusion rate) *\n");

    break;
  }

  case GETOPT: {
    OPTSPEC *os;

    next_arg(os, args, OPTSPEC *);
    os->value = (char *) &drate;
    os->set = oset;
    os->vital = -1;
    break;
  }

  case OPTVITAL: {
    int *safe;

    next_arg(safe, args, int *);
    *safe &= oset;
    break;
  }
  }
  break;
}

case VART: {
  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (vset) {
          fs = (FILESPEC *) cit[fsc].data;

          if (bprintf(fs->fd, "%K%D\n", "DRATE", drate))
            printf("write of DRATE var failed");
    }
```

```
      break;
    }

    case GETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      vs->fsc = fsc;
      vs->set = vset;
      break;
    }

    case SETVAR: {
      VARSPEC *vs;

      next_arg(vs, args, VARSPEC *);
      if (vs->fsc != NA)
            fsc = vs->fsc;

      if (vs->set != NA)
            vset = vs->set;
      break;
    }
    }
    break;
  }
  }
}



void hdl_MU(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static char fn[256];
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "MU");
      if ((ofd = fopen(fn, "w")) == NULL)
            printf("\n\nhdl_MU: Couldn't open ouput file %s\n\n", fn);
    }
    else {
      if ((ofd = fopen(fn, "a+")) == NULL)
            fprintf(stderr, "\n\nhdl_MU: Couldn't open ouput file %s\n\n", fn);
    }
    break;
  }

  case DACTVAR: {
    if (ofd)
      fclose(ofd);
    break;
  }
```

```
  case STATVAR: {
    register int j;
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *)cit[fsc].data;

      if (bprintf(fs->fd, "%K%f[]", "MU", G, MU))
            printf("write of MU var failed\n");

      for(j=0; j<G; j++)
            fprintf(ofd, "%10.10f %10.10f ", (<?= MU[j]), (>?= MU[j]));

      fprintf(ofd, "\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}




void hdl_EP(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static char fn[256];
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "EP");
      if ((ofd = fopen(fn, "w")) == NULL)
            printf("\n\nhdl_EP: Couldn't open ouput file %s\n\n", fn);
```

```
    }
    else {
      if ((ofd = fopen(fn, "a+")) == NULL)
          fprintf(stderr, "\n\nhdl_EP: Couldn't open ouput file %s\n\n", fn);
    }
    break;
  }

  case DACTVAR: {
    if (ofd)
      fclose(ofd);
    break;
  }

  case STATVAR: {
    register int j;
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *)cit[fsc].data;

      if (bprintf(fs->fd, "%K%f[]", "EP", G, EP))
          printf("write of EP var failed\n");

      for( j = 0; j < G; j++) {

          fprintf(ofd, "%10.10f %10.10f %10.10f ",
                      <?= EP[j],
                      (+= EP[j]) / LS*LS*LS,
                      >?= EP[j]);
      }
      fprintf(ofd, "\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}
```

```
void hdl_OP(int type, int action, char *args)
{
  static FILE *ofd = NULL;
  static char fn[256];
  static int set = 0;
  static int fsc = −1;

  switch(action) {

  case ACTVAR: {
    if (!ofd) {
      sprintf(fn, "%s_%s.out", cit[fsc].name, "OP");
      if ((ofd = fopen(fn, "w")) == NULL)
            printf("\n\nhdl_OP: Couldn’t open ouput file %s\n\n", fn);
    }
    else {
      if ((ofd = fopen(fn, "a+")) == NULL)
            fprintf(stderr, "\n\nhdl_OP: Couldn’t open ouput file %s\n\n", fn);
    }
    break;
  }

  case DACTVAR: {
    if (ofd)
      fclose(ofd);
    break;
  }

  case STATVAR: {
    register int j;
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *)cit[fsc].data;

    if (bprintf(fs−>fd, "%K%f[]", "OP", G, OP))
      printf("write of OP var failed\n");

      for( j = 0; j < G; j++) {

            fprintf(ofd, "%10.10f %10.10f %10.10f ",
                        <?= OP[j],
                  (+= OP[j]) / LS*LS*LS,
                        >?= OP[j]);
      }
      fprintf(ofd, "\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
```

```
    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}


void hdl_eta(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      if (bprintf(fs->fd, "%K%f[]", "ETA", G, eta))
          printf("write of ETA var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
```

```
      set = vs->set;
    break;
  }
  }
}



void hdl_lambda(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      if (bprintf(fs->fd, "%K%f[]", "LAMBDA", G, lambda))
            printf("write of LAMBDA var failed\n");
    }
    break;
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}
```

B.12 pbsort.cs

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <cm/paris.h>

#include <pmem.hs>

#include <default/pbsort.hs>


void pbsort(bool:current *a, int nel, int wid, int:current(*compar)())
{
  register int i, j;
  bool:current *tmp;
  int:current d = 0;



  with (current ) {
    tmp = (bool:current *) PMalloc(wid);

    for( i = 0; i < nel-1; i++) {

      for( j = i+1; j < nel; j++) {

            /* Where the current low is > this element */
            d = (*compar)(a + i * wid, a + j * wid);
            where(d == 1) {

              /* swap enties */
              boolcpy(tmp, a + i * wid, wid);
              boolcpy(a + i * wid, a + j * wid, wid);
              boolcpy(a + j * wid, tmp, wid);
            }
      }
    }
    PFree(tmp);
  }
}
```

B.13 slow_diffusion.cs

```
#include <default/rnd.hs>

#include <bishop.hs>
#include <model/model.hs>



#define DOIT(J, X, Y, Z) \
{ \
  *hit = (srnd(drate) == 1); \
  where (*hit == 1) { \
```

```
    send_mine(X, Y, Z, J); \
    where( *crib_count > 0) \
      send_list(X, Y, Z, J); \
  } \
}


void slow_diffusion()
{
  register int xo, yo, zo;
  register int j;


  *crib_count_cap = *crib_count;

  for (j = 0; j < G; j++) {

    where (plan[j].disp == SELL) {

      for (xo = -1; xo < 2; xo++)
            for (yo = -1; yo < 2; yo++)
              for (zo = -1; zo < 2; zo++) {
                if ((xo == 0) && (yo == 0) && (zo == 0))
                  continue;
                DOIT(j, xo, yo, zo);
              }
    }
  } /* for (j = */
}
```

B.14 get_price_info.cs

```
#include <string.h>
#include <pmem.hs>

#include <default/gcoord.hs>

#include <bishop.hs>
#include <model/model.hs>


void get_his(int xo, int yo, int zo, int g)
{
  register int j, k;
  bool:lattice f = 0;


  /* If this guy has a full crib_sheet then oh well he looses out */
  /* Also if the neighbor is not selling then there is nothing to get */
  where((*crib_count != CS) && (NMAP(plan[j].price, xo, yo, zo) == SELL)) {
    /* Search for duplicates */
    for (k=0; k < CS; k++) {
```

```
      where ((f == 0) & (*crib_count > k))
            where (crib_id[k] == MAPID(xo, yo, zo)) {
              crib_g[k*CS+g] = 1;
              f = 1;
            }

      /* No duplicates so we must add a new entry to the cribsheet */
      where ((f == 0) & (*crib_count == k)) {
            for(j = 0; j < G; j++)
              crib_g[k*CS+j] = 0;

            crib_id[k] = MAPID(xo, yo, zo);
            crib_g[k*CS+g] = 1;
            (*crib_count)++;
            f = 1;
      }

      if (&= f)
            break;
    }
  }
}

void get_sheet(int xo, int yo, int zo, int g)
{
  register int j, k, l;
  bool:lattice f = 0;

  /* If this guy has a full crib_sheet then oh well he looses out */
  where(*crib_count != CS) {

    /* l indexes the cribsheet neighbor's cribsheet */
    for (l = 0; l < (>?= *crib_count_cap); l++) {

      where (*crib_count_cap > l) {

            /* Search for duplicates or the next free location */
            /* k indexes my cribsheet */
            for (k=0; k < (>?= *crib_count); k++) {

              /* haven't reached the end of my cribsheet */
              where ((f == 0) & (*crib_count > k))

                /* is this a duplicate entry ? if yes then merge with mine */
                where (crib_id[k] == NMAP(crib_id[l], xo, yo, zo)) {
                  crib_g[k*CS+g] |= NMAP(crib_g[l*CS+g], xo, yo, zo);
                  f = 1;
                }

              /* I have reached the end of my cribsheet */
              /* and need to add a new entry */
```

```
        where ((f == 0) & (*crib_count == k)) {

          /* init the new entry's flags to zero */
          for(j = 0; j < G; j++)
            crib_g[k*CS+j] = 0;

          /* copy the neighbor's id and flag for this good */
          crib_id[k] = NMAP(crib_id[l], xo, yo, zo);
          crib_g[k*CS+g] = NMAP(crib_g[l*CS+g], xo, yo, zo);

          /* increment my cribsheet entry  count */
          (*crib_count)++;
          f = 1;
        } /* where ((f == 0) & (*crib_count == k)) */

      } /* for (k=0; k < (>?= *crib_count); k++) */

    } /* where (*crib_count_cap > l) */

  } /* for (l = 0; l < (>?= *crib_count_cap); l++) */

 } /* where(*crib_count != CS) */
}
```

B.15 gc_cribs.cs

```
#include <pmem.hs>

#include <default/gcoord.hs>

#include <bishop.hs>
#include <model/model.hs>



void gc_cribs()
{
 register int i, j;
 bool:lattice b;
 int:lattice x, y, z;

 everywhere {

  /* Search cribsheets and eliminate entries for changed prices */
  for (i = 0; i < CS; i++) {

    b = 0;
    where (crib_id[i] >= 0)
       if (CM_global_count_context() != 0) {

         g_coord(crib_id[i], &x, &y, &z, LS);

         for (j = 0; j < G; j++) {
```

```
          where (crib_g[i*CS+j] & [x][y][z]plan[j].chng)
            crib_g[i*CS+j] = 0;

          b |= crib_g[i*CS+j];
        }

        where (b == 0)
          crib_id[i] = NO_ONE;
      }
  }


  for(i=0; i<G; i++)
    plan[i].chng = 0;


  /* Search cribsheets and eliminate gaps */
  x = *crib_count − 1;

  for (i=0; i < CS; i++)

    where (*crib_count > i)
          if (CM_global_count_context() != 0)

            where(crib_id[i] < 0)
              if (CM_global_count_context() != 0) {

                crib_id[i] = crib_id[x];
                for (j = 0; j < G; j++)
                      crib_g[i*CS+j] = crib_g[x*CS+j];
                (*crib_count)−−;
              }
  }
}
```

B.16 init_util.cs

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/time.h>
#include <time.h>
#include <varargs.h>

#include <bishop.hs>
#include <default/command.h>
#include <default/init_util.hs>



static int i1 = 0, i2 = 0;

static TABLE utils_extensions[] = {
  { "TIME", 0, VART, hdl_time, NULL },
  { NULL, NULL, NULL, NULL, NULL }
```

```
};

int utilexts[5];


int wrtlog(fi, ch, a, b, c, d, e, f, g, h, i, j)
     FILE *fi;
     char *ch;
     double a, b, c, d, e, f, g, h, i, j;
{
  int ii;
  ii = fprintf(fi, ch, a, b, c, d, e, f, g, h, i, j);
  return(ii);
}


void init_util(int argc, char *argv[], int step)
{
  switch(step) {

  case PRELIM: {

    if (!i1) {

      /* Register utility variables */
      reg_spec(utils_extensions, utilexts);
      i1 = -1;
    }
    break;
  }

  case FINAL:{

    if (!i2) {
      i2 = -1;
    }
    break;
  }
  }
}




void finish_util()
{
  if (i2)
    {
      i2 = 0;
    }
}



struct tm *tm_p()
{
  static time_t t;
  t=time(NULL);
  return(localtime(&t));
}
```

```
void hdl_time(int type, int action, char *args)
{
  static int set = 0;
  static int fsc = -1;
  char tim[1024];
  char msg[1024];
  char *p;

  switch(action) {

  case STATVAR: {
    FILESPEC *fs;

    if (set) {
      fs = (FILESPEC *) cit[fsc].data;

      next_arg(p, args, char *);
      strftime(tim, 1024, "%c", tm_p());

      sprintf(msg, "%s:  %s\n", tim, p);

      if (bprintf(fs->fd, "%K%S", "TIME", msg))
          printf("write of TIME var failed\n");
      break;
    }
  }

  case GETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);

    vs->fsc = fsc;
    vs->set = set;
    break;
  }

  case SETVAR: {
    VARSPEC *vs;

    next_arg(vs, args, VARSPEC *);
    if (vs->fsc != NA)
      fsc = vs->fsc;

    if (vs->set != NA)
      set = vs->set;
    break;
  }
  }
}
```

B.17 gcoord.cs

```
#include <default/gcoord.hs>


void g_coord(int ai, int *x, int *y, int *z, int s)
{


  if (ai < s * s)
    *z = 0;
  else {
    *z = (int) (ai / (s * s));
    ai -= *z * s * s;
  }

  if (ai < s)
    *y = 0;
  else {
    *y = (int) (ai / s);
    ai -= *y * s;
  }

  *x = ai;
}


int a_id(int x, int y, int z, int s)
{
  return(x + y * s + z * s * s);
}




void g_coord(int:current ai, int:current *x, int:current *y, int:current *z, int s)
{

  where (ai < s * s)
    *z = 0;
  else {
    *z = ai / (s * s);
    ai -= *z * s * s;
  }

  where (ai < s)
    *y = 0;
  else {
    *y = ai / s;
    ai -= *y * s;
  }

  *x = ai;
}
```

```
int:current a_id(int:current *x, int:current *y, int:current *z, int s)
{
  return(*x + *y * s + *z * s * s);
}
```

B.18 rnd.cs

```
#include <cm/paris.h>
#include <stdlib.h>

#include <default/rnd.hs>

short:current srnd(short n)
{
  static short seeded = 0;


  if (!seeded)
    psrand( (seeded = getuid()+time(NULL)) );

  with (current) {
    return( (short:current) (1 + (prand() % n)) );
  }
}


int:current rnd(int n)
{
  static int seeded = 0;


  if (!seeded)
    psrand( (seeded = getuid()+time(NULL)) );

  return(1 + (prand() % n));
}


float:current frnd(float n)
{
  float:current r;

  with (current) {
    CM_f_random_1L(&r, 23, 8);
    r *= n;
    r += 1.0F;
    return(r);
  }
}



float:current frnd(void)
```

```
{
  float:current r;

  with (current) {
    CM_f_random_1L(&r, 23, 8);
    return(r);
  }
}
```

B.19 misc.cs

```
#include <bishop.hs>


void death(int s)
{
  finish_util();
  finish_model();
  finish_bishop();
  finish_command();
  exit(s);
}


wait_for_key(int i)
{
  printf("\nFinished period %d\n", i);
  getchar();
}
```

B.20 pmem.cs

```
#include <cm/paris.h>
#include "pmem.hs"



#define CMMIL CM_maximum_integer_length

static void:void *ptable[MAXPNTRS];
static short plen[MAXPNTRS];
static int init = 0;


void add_to_table(void:void *p, int l)
{
  register int i;
```

```
  for(i=0; i<MAXPNTRS; i++)
    if (plen[i] < 0) {
      plen[i] = l;
      ptable[i] = p;
      return;
    }

  fprintf(stderr, "\n\nParallel Memory Package:    No more pointers \n\n");
  exit(-1);
}



int table_loc(void:void *p)
{
  register int i;

  for(i=0; i<MAXPNTRS; i++)
    if (ptable[i] == p)
      return(i);

  return(-1);
}


void PCopy(void:void *source, void:void *dest, int length)
{
  boolcpy(dest, source, length);
}


void PCopy(void *source, void:void *dest, int length)
{
  register int i, bl;
  unsigned int s;
  register int sl = sizeof(s);
  unsigned char *src = (unsigned char *) source;


  bl = length / 8;
  bzero(&s, sl);

  if (length < sl) {
    bcopy(src, &s, bl);
    CM_u_move_constant_1L(dest, s, length);
  }

  else {
    for (i = 0; i < bl; i += sl) {

      bcopy((src+i), &s, sl);
      CM_u_move_constant_1L(dest, s, sl*8);
      dest = CM_add_offset_to_field_id(dest, sl*8);
    }
  }
}
```

```
void PZero(void:void *dest)
{
  register int i, l;

  if ((l = table_loc(dest)) < 0) {
    fprintf(stderr, "\n\nPZero:  this pointer is invalid\n\n");
    exit(-1);
  }

  if (plen[l] < CMMIL)
    CM_u_move_zero_1L(dest, plen[l]);

  else {
    for (i = plen[l]; i > CMMIL; i -= CMMIL) {

      CM_u_move_zero_1L(dest, CMMIL);
      dest = CM_add_offset_to_field_id(dest, CMMIL);
    }
    CM_u_move_zero_1L(dest, i);
  }
}



void PZero(void:void *dest, int bits)
{
  register int i;

  if (bits < CMMIL)
    CM_u_move_zero_1L(dest, bits);

  else {
    for (i = bits; i > CMMIL; i -= CMMIL) {

      CM_u_move_zero_1L(dest, CMMIL);
      dest = CM_add_offset_to_field_id(dest, CMMIL);
    }
    CM_u_move_zero_1L(dest, i);
  }
}



void PMemInit()
{
  register int i;

  if (init == 0) {
    for(i=0; i<MAXPNTRS; i++) {
      ptable[i] = NULL;
      plen[i] = -1;
    }
    init = 1;
  }
}
```

```
void:void *PCalloc(shape s, int num, int size)
{
  void:void *t;

  with (current) {
    everywhere {

      if (( t = palloc(s, size*num)) == NULL) {

            fprintf(stderr, "\n\nPCalloc failed\n\n");
            exit(-1);
      }

      add_to_table(t, size);
      PZero(t);
      return(t);
    }
  }
}


void:void *PCalloc(int num, int size)
{
  void:void *t;

  with (current) {
    everywhere {

      if (( t = palloc(current, size*num)) == NULL) {

            fprintf(stderr, "\n\nPCalloc failed\n\n");
            exit(-1);
      }

      add_to_table(t, size);
      PZero(t);
      return(t);
    }
  }
}


void:void *PMalloc(shape s, int size)
{
  void:void *t;

  with (current) {
    everywhere {

      if (( t = palloc(s, size)) == NULL) {

            fprintf(stderr, "\n\nPMalloc failed\n\n");
            exit(-1);
      }

      add_to_table(t, size);
      return(t);
    }
  }
```

```
}


void:void *PMalloc(int size)
{
  void:void *t;

  with (current) {
    everywhere {

      if (( t = palloc(current, size)) == NULL) {

            fprintf(stderr, "\n\nPMalloc failed\n\n");
            exit(-1);
      }

      add_to_table(t, size);
      return(t);
    }
  }
}


void:void *PRealloc(shape s, void:void *p, int size)
{
  void:void *t;
  int l;

  with (current) {
    everywhere {

      if ((l = table_loc(t)) < 0) {

            fprintf(stderr, "\n\nPRealloc:  this pointer is invalid\n\n");
            exit(-1);
      }

      t = PMalloc(s, size);

      if (plen[l] > size)
            PCopy(p, t, size);
      else
            PCopy(p, t, plen[l]);

      PFree(p);
      ptable[l] = t;
      plen[l] = size;
      return(t);
    }
  }
}


void:void *PRealloc(void:void *p, int size)
{
  void:void *t;
  int l;

  with (current) {
```

```
    everywhere {

      if ((l = table_loc(t)) < 0) {

            fprintf(stderr, "\n\nPRealloc:  this pointer is invalid\n\n");
            exit(−1);
      }

      t = PMalloc(size);

      if (plen[l] > size)
            PCopy(p, t, size);
      else
            PCopy(p, t, plen[l]);

      PFree(p);
      ptable[l] = t;
      plen[l] = size;
      return(t);
    }
  }
}


void PFree(void:void *p)
{
  int l;

  with (current) {
    everywhere {

      if ((l = table_loc(p)) < 0) {

            fprintf(stderr, "\n\nPFree:  this pointer is invalid\n\n");
            exit(−1);
      }

      pfree(p);
      plen[l] = −1;
      ptable[l] = NULL;
    }
  }
}
```

Appendix C

Symbol Definitions

There are a number of typesetting conventions used throughout this thesis. Mathematics symbols are typeset in *script*; C and C* variables, statements and code fragments are typeset in `typewriter`; names of concepts which we introduce and could be misunderstood as something else are typeset in SMALL CAPS. Below is a table defining the various symbols used.

λ_{ij}	randomly selected weight for agent i and good j.
Δ_{ij}	agent i's desire for good j.
ap_j^H	highest advertised price seen for good j during the current period.
ap_j^L	lowest advertised price seen for good j during the current period.
p_j^H	highest price paid for good j, if the agent is a customer of good j, or received for good j, if the agent is a merchant of good j, during the current period.
p_j^L	lowest price paid for good j, if the agent is a customer of good j, or received for good j, if the agent is a merchant of good j, during the current period.

Table C.1: Mathematical Symbols

$\mathcal{LS}$	the size of the CA lattice. The number of agents in the lattice is LS^3.
$\mathcal{G}$	the number of goods in the economy, indexed by j.
y_{ij}^t	money that agent i holds at time t.
X_{ij}^t	holdings of agent i in good j at time t.
ρ	the rate of information diffusion. Prices move into the neighborhood with a probability of $\frac{1}{\rho}$.
$\vec{Q}_i^t$	vector of planned acquisitions of agent i at time t.
$\vec{R}_i^t$	vector of planned sales of agent i at time t.
$U_i^t(X_{ij}^t)$	utility for agent i at time t.
$MU \stackrel{\text{def}}{=} \frac{\partial U}{\partial X}$	marginal utility for agent i at time t.
$E_i^t[p_j]$	expected market price of good j for agent i at time t.
P_{ij}^{*t}	optimal selling price of good j for agent i at time t.
$\tilde{P}_{ij}^t$	the lowest price gathered from agents i's neighborhood during period t for good j.
${}^{paid}P_{ij}^t$	highest price paid for good j (either by agent i to a merchant or by a customer to agent i) up to period t.
${}_{paid}P_{ij}^t$	lowest price paid for good j (either by agent i to a merchant or by a customer to agent i) up to period t.
${}^{seen}P_{ij}^t$	highest price seen for good j by agent i up to period t.
${}_{seen}P_{ij}^t$	lowest price seen for good j by agent i up to period t.

References

[1] P. Albin and D. Foley. Decentralized, dispersed exchange, without an auctioneer: A simulation study. CUNY Working Paper, December 1988.

[2] B.M. Boghosian, W. Taylor IV, and D.H. Rothman. A cellular automata simulation of two-phase flow on the cm-2 connection machine computer. *Proceedings of Supercomputing '88, IEEE, Kissimmee, FL*, November 1988. November 14-18.

[3] G.R. Butters. Equilibrium distributions of sales and advertising prices. *Review of Economic Studies*, 44, 1977.

[4] Doyne Farmer, Tommaso Toffoli, and Stephen Wolfram, editors. *Cellular Automata: Proceedings of an Interdisciplinary Workshop*, Amsterdam: North-Holland, 1984. Elsevier Science Publishing Co. Los Alamos, New Mexico, March 7-11, 1983.

[5] Martin Gardner. The fantastic combinations of john conway's new solitaire game 'life'. *Scientific American*, 223(4):120–123, April 1970.

[6] Thinking Machines Corporation, Cambridge, Massachussetts. *C* Programmer's Guide, Version 6.0.2*, june 1991.

[7] Tommaso Toffoli and Norman Margolus. *Cellular Automata Machines: A New Environment for Modeling.* MIT Press, Cabridge, Mass., 1989.

[8] Hal R. Varian. *Microeconomic Analysis.* W.W. Norton Press., 2nd edition, 1984. Chapter 3. ISBN 0-393-95282-7.

[9] John von Neumann. *Theory of Self-Reproducing Automata.* Univ. of Illinois Press, 1966. Edited and completed by Arthur Burks.