PARALLELIZATION OF ENTITY-BASED MODELS IN COMPUTATIONAL SOCIAL SCIENCE: A HARDWARE PERSPECTIVE

by

Dale K. Brearcliffe

A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
in Partial Fulfillment of
The Requirements for the Degree
of
Master of Arts
Interdisciplinary Studies

Committee:

_____ Director

_____

_____

_____ Program Director

_____ Dean, College of Humanities and Social Sciences

Date: _____ Fall Semester 2017
George Mason University
Fairfax, VA

Parallelization of Entity-Based Models in Computational Social Science: A Hardware Perspective

A Thesis submitted in partial fulfillment of the requirements for the degree of Master of Arts at George Mason University

by

Dale K. Brearcliffe
Bachelor of Science
California State University Hayward (East Bay), 1983

Director: Andrew Crooks, Associate Professor;
Department of Computational and Data Sciences

Fall Semester 2017
George Mason University
Fairfax, Virginia

DEDICATION

To my grandmother, Mary Eleanor Brearcliffe (1914 - 2009), who understood the importance of education.

ACKNOWLEDGMENTS

I thank my advisor, Dr. Andrew Crooks, and thesis committee members Dr. Robert Axtell and Dr. William G. Kennedy for insightful instruction and hours of enjoyable conversation. I also thank Karen Underwood for herding all the cats in the same direction.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

Agent-Based Model ..... ABM
Application Programming Interface ..... API
Application Specific Integrated Circuit ..... ASIC
Cellular Automata ..... CA
Central Processing Unit ..... CPU
Computational Social Science ..... CSS
Compute Unified Device Architecture ..... CUDA
Entity-Based Model ..... EBM
General Purpose Graphics Processing Unit ..... GPGPU
GNU Compiler Collection ..... GCC
GNU's Not Unix (A recursive acronym) ..... GNU
Graphics Processing Unit ..... GPU
High Performance Computing ..... HPC
Individual-Based Model ..... IBM
Integrated Circuit IC
Logical Processing Element ..... LPE
Multiple Instruction Multiple Data ..... MIMD
Multiple Instruction Streams Single Data Stream ..... MISD
Network on Chip ..... NoC
NVIDIA CUDA Compiler ..... NVCC
Reduced Instruction Set Computing ..... RISC
Single Instruction Multiple Data ..... SIMD
Single Instruction Stream Single Data Stream ..... SISD
Single Program Multiple Data ..... SPMD
System on Chip ..... SOC
Yet Another Resource Negotiator ..... YARN
Zero-Intelligence Traders ..... ZIT

ABSTRACT

PARALLELIZATION OF ENTITY-BASED MODELS IN COMPUTATIONAL SOCIAL SCIENCE: A HARDWARE PERSPECTIVE

Dale K. Brearcliffe, MAIS

George Mason University, 2017

Thesis Director: Dr. Andrew Crooks

The use of simulations by social scientists in exploring theories and hypotheses is well documented. As computer systems have grown in capacity, so have interests of social scientists in executing larger simulations. Social scientists often approach their simulation design from the top down by selecting an Entity-Based Model (**EBM**) framework from those that are readily available, thus limiting modeling capability to the available frameworks. Ultimately, the framework is dependent upon what is at the bottom, the hardware architecture that serves as the foundation of the computing system. Parallel hardware architecture supports the simultaneous execution of a problem split into multiple pieces. Thus, the problem is solved faster in parallel. In this thesis, a selection of parallel hardware architectures is examined with a goal of providing support for EBMs. The hardware's capability to support parallelization of EBMs is described and contrasted. A simple EBM is tested to illustrate these capabilities and implementation challenges specific to parallel hardware are explored. The results of this research offer

social scientists better informed choices than the sequential EBM frameworks that currently exist. Matching the model to the correct supporting hardware will permit larger scale problems to be examined and expands the range of models that a social scientist can explore.

1. INTRODUCTION

1.1. Introduction

Computational Social Science (**CSS**) is an interdisciplinary (Meeth, 1978) field in which mathematics is used computationally to explore social science questions and answer social science problems. CSS encompasses a number of computational approaches of which one is modeling (Cioffi-Revilla, 2014, p. 2). Agent-based models (**ABM**) and individual-based models (**IBM**) are the predominant modeling systems used by CSS. The description is used despite attempts to redefine CSS by others who seem unfamiliar with the existing field. One such attempt is a brief 2009 article in Science magazine where fifteen authors argue for the "emergence of a data-driven 'computational social science'" without mentioning any form of agent-based modeling or acknowledging previous definitions of CSS (Lazer, et al., 2009). A better description of CSS that encompasses data science and scientific algorithms was provided by Benthall (2016) who writes that scientific algorithms "implement statistical inference" and views "computational social science as the application of scientific algorithms to understand social phenomena." This view should be construed as including agent-based models as scientific algorithms even though such models were not explicitly mentioned. Watts (2013) is explicit in describing modeling as an integral part of CSS.

ABMs are a means to model issues involving people's decision-making or behaviors whereas IBMs model issues generally focused on animals and plants. These terms are often used interchangeably (Railsback & Grimm, 2012, pp. xi-xii). There is sufficient overlap in tools, use and labeling between these model types that I will use the phrase Entity-Based Models (**EBM**) to describe both (Cleary, Smith, Vassilevska, & Jefferson, 2005). This use is not to be confused with "Equation-Based Modeling", another definition of EBM, where such simulations are described by a set of equations that are evaluated during model execution (Parunak, Savit, & Riolo, 1998). The use of EBM in the context of this thesis is for the convenience of the reader and not an attempt at redefinition.

CSS in application has focused on the social science aspect of this interdisciplinary field. There are over a thousand papers and EBMs available from academic sources such as the Journal of Artificial Societies and Social Simulation (http://jasss.soc.surrey.ac.uk/), the OpenABM Consortium (https://www.openabm.org/) and International Conference on Social Computing, Behavioral-Cultural Modeling, & Prediction and Behavior Representation in Modeling and Simulation (http://sbp-brims.org) describing results of the application of an EBM to a social problem or exploring some other area. Noticeably less is the attention paid to the computer science and engineering aspect of CSS. Yet without computers, social scientists might be forced, as was Schelling (1969), to conduct simulations in a single dimension with pen and paper. For CSS to be interdisciplinary, the contributions of computer science and computer engineering must be embraced equally with social science. Rather than leave

this solely to the availability of computer scientists, social scientists in the CSS field should be fully engaged with the development of the tools they rely on, instead of merely users of existing EBM software frameworks that is commonly the case today. Frameworks will be discussed in Section 2.3.

The use of EBMs in science has grown to embrace a number of fields to include, but not limited to, computational biology (e.g. Walker, Hill, Wood, Smallwod, & Southgate, 2004), public safety (e.g. Ren, Yang, & Jin, 2009), public health (e.g. Crooks & Hailegiorgis, 2014), economics (e.g. Axtell, 2008), and ecology (e.g. Grimm & Railsback, 2005). Unfortunately, EBMs are often restricted by the slow computing speed and limited memory of the computer hardware they operate on requiring model alteration (Wendel & Dibble, 2007). Some models compensate for this by limiting the number of agents (e.g. Šalamon, 2011, p. 122; Wilensky & Rand, 2015, pp. 418-419), the space in which the agents operate (e.g. Wilensky & Rand, 2015, pp. 418-419), or amount of time in which computation occurs (e.g. Hogeweg & Hesper, 1983; Heijnen, Chappin, & Nikolic, 2014). The increase in computing power as described by Moore's law (1965) has enabled social scientists to scale their simulations in stride. Unfortunately, Moore's law may not match the needs of social scientists as some may wish to model entire populations (e.g. Axtell, 2016).

Hardware limitations, primarily heat dissipation, have prevented computer processors from continuing their increase in computational power. In a bid to overcome these limitations, computer hardware engineers have created processors with multiple

processing **cores** (multi-core central processing units (**CPU**)), general purpose graphics processing units (**GPGPU**), specialized processors (application specific integrated circuit (**ASIC**) system on **chip** (**SOC)** coprocessors), and networked computing (homogeneous and heterogeneous clusters). These architectural advances have led to an era of "Big Data" with the expectation that large amounts of data may be processed. The large data boundary, whose processing exceeds either computational or temporal constraints, is monotonically increasing. These advances offer the possibility of EBMs with software elements that operate simultaneously, in parallel to each other. Existing EBM software application frameworks such as NetLogo (Wilensky, 1999), intended for the single (sequential) processor, were not designed to take advantage of this type of hardware architecture. The demands by some social scientists for large-scale simulations have outpaced the computational power available to these sequential frameworks (e.g. Hayes, et al., 2014; Xiong, 2015). Efforts in creating frameworks that take advantage of these parallel processing architectures, such as Flexible Large-scale Agent Modelling Environment (FLAME) for heterogeneous computing (Holcombe, Coakley, & Smallwood, 2006) or **GPGPU** based models (Lysenko & D'Souza, 2008), have fragmented the approach to EBMs across these environments. In some cases, it may be necessary to include a computer scientist as part of the modeling effort.

Social scientists usually approach their simulation design from the top down by selecting an EBM framework from those that are readily available such as NetLogo. This approach limits the social scientist to the capability of the chosen framework. Ultimately, the framework is dependent upon what is at the bottom, the hardware that serves as the

foundation of the computing system. If a sequential framework such as NetLogo is chosen, it will fail to take full advantage of a modern hardware architecture foundation.

1.2. Research Questions

In this thesis, I approach the issue from the bottom up by examining a selection of parallel hardware architectures from the perspective of providing support for **EBM**s. This thesis attempts to answer three research questions: 1) Does the underlying hardware play a role in the social scientist's capability to create large-scale models? 2) If so, does the hardware change the approach and skills needed for modeling? 3) Is it worth the effort?

1.3. Thesis Outline

The next chapter will provide background information on it **EBM**s, why parallel models are needed, some existing sequential EBM frameworks, hardware architecture categories, and some recent efforts to parallelize EBM. Chapter 3 describes the general methodology and the EBM selected for experimentation. Applied parallelism is the focus of Chapter 4. First, a description of a previous approach using a multi-core **CPU** is described. This is followed by a description of the experiment on each of the **GPGPU**, **ASIC**, and High Performance Computing (**HPC**) architectures. Chapter **Error! Reference source not found.** discusses model verification and validation. Finally, Chapter 6 provides a summary of the results and conclusion, along with areas of future work.

2. BACKGROUND

2.1. Entity-Based Models

The label Entity-Based Model (**EBM**) is used as an amalgamation of agent-based models (**ABM**) and individual-based models (**IBM**) (Cleary, Smith, Vassilevska, & Jefferson, 2005). ABMs are often used to model people (agents) operating in a simulated environment using a small number of rules to guide the actions of the agents and changes to the environment. Within this *in silico* laboratory, societies emerge whose interactions can be studied over time and by varying the conditions in which they exist. Epstein and Axtell (1997) described this as "generative social science" because it encompasses neither deductive nor inductive reasoning. Instead, a new scientific process, the "artificial society" is used as an instrument of exploration. In a similar fashion, IBMs are generally used to model animals and plants in a simulated ecological environment. Classic mathematical ecology was based on the idea of the homogeneous averaged individual. IBMs introduced heterogeneous individuals with complex lifecycles and individual changing resources. Rather than showing mathematically stable ecological systems, IBMs permit the study of local changes in population and resources from which the ecological system as a whole is an emergent property (Uchmański & Grimm, 1996).

The acceptance of EBMs in social science has had a number of impacts. EBMs as computational models can be viewed as the manifestation of a "third way" of expressing

the ideas of social science, going beyond natural language and mathematics (Ostrom, 1988; Gilbert & Terna, 2000). As an investigative science tool, it has removed some of the restrictions imposed by methods that used strict mathematical modeling and proofs (Bankes, 2002). EBMs can be used to explore and sometimes explain emergent behavior that occurs when the microscopic behavior of local agent interactions create macroscopic patterns across the entire artificial society (Epstein, 1999). Here "emergent" is used as a placeholder to acknowledge a process that cannot currently be explained but one day might be. Finally, EBMs permit social theories to be tested and experimented on in a controlled environment (Macy & Willer, 2002). These models can be run hundreds of times or more before analyzing the collective results.

In this thesis, a third type of modeling approach, the cellular automata (**CA**), is regarded as a limited type of EBM and are thus subsumed by them. CAs are one of the simplest types of social simulation models (Cioffi-Revilla, 2014, pp. 16-17, 231-232). The key difference is that there are no agents in a CA to move within the environment, although there can be an illusion of movement as stationary regions react to changes in their surrounding environment (Gardner, 1970). Not all EBMs have agents that move and the difference between CA and EBMs become less distinct as the two modeling approaches are combined in a single model (Crooks, 2017).

2.2. The Need for Parallelization

Herbert Simon (1955; 1996, p. 166) described "bounded rationality" as a limit on the ability of an adaptive system to consider all choices in a complex environment. The

adaptive system he referred to was a person. In a similar manner, computational limitations create a bounded rationality for an **EBM** software framework when the desired number of agents, the size of the environment, or the complexity of calculations cannot be concluded within a desired temporal period (Papadimitriou & Yannakakis, 1994; Tsang & Martinez-Jaramillo, 2004).

Computer programs have historically been designed to execute sequentially on a single computing device. Computational limitations can affect the ability of a social scientist to create the simulation needed for a social issue to be fully explored. For example in one financial simulation, the authors were able to reproduce a model that contained liquidity inconsistencies attributed to its one-thirty-second scale, but were still limited to a one-quarter scale (Hayes, et al., 2014). In a livestock EBM, the agents needed to be made partially homogeneous by clustering them into heterogeneous, representative herds (Bradhurst, Roche, East, Kwan, & Garner, 2016). Geographic information systems can capture spatial data at a level of resolution that exceeds the computational ability to model at the same resolution (Crooks, Castle, & Batty, 2008). Computational complexity changes from tractable to intractable when agent and environment interactions move from a polynomial scale to an exponential scale (Cioffi-Revilla, 2014, p. 64).

Bounded rationality can mitigate computational limitations by decreasing the number of interactions between agents and their environment. Doing so decreases the number of calculations and the amount of time needed to complete the model. For large-scale models, this may not be enough. Increasing the number of calculations that can take

place during each time step of the model can increase the model run time. For these large-scale models, parallelization offers a potential path to success.

2.3. Frameworks for Entity-Based Models

An **EBM**'s existence is predicated on more than just the hardware that serves as its foundation as shown in Figure 1. Other intermediate layers also support the EBM and one of the more important in this stack is the EBM framework. These frameworks fall into two types: Either a general framework useful for creating a wide range of simulations or a framework designed specifically for a single model. Examples of general EBM frameworks are NetLogo, Swarm (Swarm Development Group, 1999), Repast (North, et al., 2013) and Multi-Agent Simulator Of Neighborhoods (MASON) (Luke, Cioffi-Revilla, Panait, Sullivan, & Balan, 2005). Designed EMB frameworks are specific to the problem a social scientist is exploring and so can be as numerous as there are social scientists. The designed EBM is built using a general-purpose programming language in lieu of a general EBM framework. An example of this is the Sugarscape model, created with the programming languages Object Pascal and C (Axtell, Axelrod, Epstein, & Cohen, 1996). Whether general or designed, each EBM framework is based upon one of many available general programming languages. While the EBM framework is the layer most visible to the social scientist, it is not the intention to study these frameworks but merely to acknowledge them for context. Section 2.5 provides use cases of designed and general frameworks.

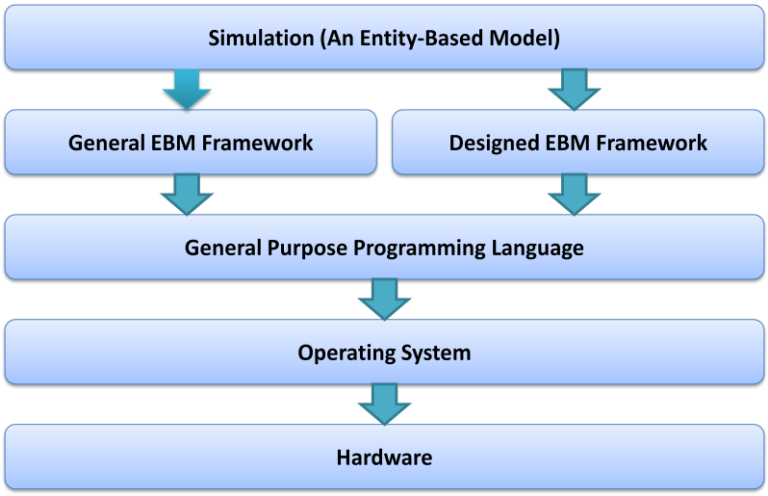

Figure 1 - An Abstraction of an Entity-Base Model's Relationship to Hardware

2.4. Hardware Categories

There are a number of approaches to achieving parallelization. I have categorized these approaches from a hardware perspective based on the multi-core central processing unit (**CPU**), graphic processing unit (**GPU**) also referred to as a general purpose graphic processing unit (**GPGPU)**, application specific integrated circuit (**ASIC**) coprocessors, homogeneous clusters, and heterogeneous clusters as shown in Table 1. Although these categories are a generalization and could be sub-categorized, they are considered sufficient for this thesis.

Table 1 - Hardware Categories

Hardware Category	Description
Multi-Core Central Processing Unit	A CPU with more than one logical processing element.
Graphic Processing Unit	Composed of thousands of logical processing elements. These LPEs do not communicate with each other, are simpler, and less capable individually than those found in CPUs.
Application Specific Integrated Circuit	Purposely designed for some function. Can contain multiple highly specialized RISC cores that operate independently or collectively on internal NoCs.
Homogeneous Computing Nodes	Nodes of similarly designed and configured multi-core computers physically located together and optimized for high-speed communication.
Heterogeneous Computing Nodes	A distributed network of nodes different types of computer hardware and operating systems. They can be geographically distributed, communicating across the Internet.

The first category relies on modern personal computers (servers, desktops, and laptops) that have central processing units with multiple **logical processing elements** (**LPE**) commonly referred to as **cores**. While servers are not commonly thought of as a personal computer, there are but small differences between them and the hardware architecture of desktops and laptops. The LPE label is used to differentiate from a *physical* processing element that may contain one or more *logical* elements. Typically, there can be two *logical* processing elements per *physical* element often referred to as hyper-threading. Software applications count the number of LPEs when determining the number of available cores. Also, note that multiple *physical* processing elements can be found within a single **integrated circuit** (**IC**). This IC can also be referred to as a CPU or

socket. The number of LPEs in a CPU is currently about ten to twenty for desktops and laptops, or near one hundred for a server. If past practices are a prediction of the future, hardware companies will increase the number of LPEs. An **EBM** can be divided among these LPEs so that its sub-processes execute in parallel to each other rather than sequentially (Herlihy & Shavit, 2012). A second category is the addition of one or more graphic processing units to the computer. Each GPU is composed of thousands of LPEs. These LPEs do not communicate with each other, are simpler, and less capable individually than those found in CPUs, but the total number of them, operating in parallel to each other, allow for a large increase in computing power per unit of time (Owens, et al., 2008; Dematte & Prandi, 2010). A third category is the use of ASIC coprocessors with a CPU. These are purposely designed to implement a function in hardware. They can contain multiple highly specialized reduced instruction set computing (**RISC**) **cores** that operate independently or collectively on their own internal network on **chip** (**NoC**) optimized for efficient high-speed communication (Richie, Ross, Park, & Shires, 2015; Olofsson, 2016). Section 2.5 provides some examples of EBMs using these architectures.

These first three categories are often attained locally on a single computer that can be directly controlled by the social scientist. The next two categories could require the social scientist to rely on systems outside their direct control. The fourth category is a network of homogeneous computing nodes. These nodes are often multi-core computers in of themselves that are physically located together and optimized for high-speed communication. Each node has a similar hardware configuration and identical operating systems (Geist & Reed, 2015). The fifth and final category is a distributed network of

heterogeneous computing nodes. These nodes are each treated as a single computing entity where each can be a different type of computer hardware and operating system. They are geographically distributed, communicating across the Internet (Anderson, 2004). These categories are not strictly delimited and can be blended together creating a hybrid system. In fact, all personal computers must have a CPU. The addition of GPU or ASIC hardware is optional. Similarly, a homogeneous computing network is usually composed of computing nodes with multi-core CPUs. A heterogeneous computing network can be composed of any personal computer configuration of CPUs, GPUs and ASIC.

2.5. Recent Efforts to Parallelize EBMs or Conduct Experiments

Social scientists have created parallel simulations and conducted case studies to experiment with the feasibility of parallel **EBM**s. One of the most obvious ways to accomplish parallelization is to take advantage of the multi-core **CPU**. This resource is already on the social scientist's computer and under their direct control. A predator-prey model implemented by Fachada, Lopes, Martins, and Rosa (2016) on a computer with twelve **LPE**s obtained a speed increase forty times greater than its single LPE equivalent. A price in increased programming complexity was paid, however, as the single LPE version used NetLogo, a simple EBM framework used to teach modeling, while the multi-core model was created in the general-purpose computing language Java. Gong, Tang, Bennett, and Thill (2012) showed spatial EBMs, those using a two or more dimensional grid on which agents interact, benefited from a speed increase as more cores

are added. However when the spatial size was increased along with the cores, the **speed multiplier** was diminished, demonstrating a lack of efficiency.

GPUs can be added to the social scientist's computer if it does not already have one. These units add thousands of LPEs to the system, leading some to label them many-core processors, but at the cost of greater programming difficulty and reduced capability for each GPU core. Leinweber, et al. (2014) used a GPU for an EBM of yeast flocculation that resulted in a speed increase of 736 times over the single LPE version for 20,000 agents. The speed increase was sufficient to permit real-time observation of the model in three dimensions. Another three dimensional visualization by Husselmann and Hawick (2011) was achieved in modeling the flocking behavior of "Boids." Although Reynolds' (1987) original EBM was only two dimensional, the GPUs allowed for the third dimension, an increase in the number of agents, and an extension to observe multiple competing flocks within the same simulation. Several GPU boards were compared with the best results showing about 33,000 agents could be processed in a single time step of 0.06 seconds.

Laville, Mazouzi, Lang, Philippe, and Marilleau (2013) created an EBM of soil science that combined the CPU and GPU. By assigning complicated agents to the CPU and simple agents or the spatial environment to the GPU, each type of processing unit was able to maximize its unique capability. The EBM was incrementally modified from its original sequential algorithm to one that could be supported by the CPU and GPU. In the ultimate configuration for a hybrid CPU/GPU, some of the newest processors have

the CPU and GPU on the same **chip**. This promises tighter integration and sharing of memory resources. Wang, Rubin, Wu, and Yalamanchili (2013) used one of these new processors for an EBM traffic simulation of two million vehicles. They showed a speed increase of thirty-four times compared to a CPU alone.

No EBMs using an ASIC coprocessor have been found to be published based on an extensive literature review search. However, in a case study of a **CA** model using an ASIC coprocessor, Aaberge (2004) achieved a speed multiplier of sixteen times compared to using only the main CPU. It was noted that the coprocessor was capable of multiple-instruction, multiple-data (**MIMD**) and single-instruction, multiple-data (**SIMD**) tasks. **MIMD** and **SIMD** are hardware techniques useful for parallelizing data processing as categorized by Flynn (1966).

HPC clusters offer the possibility of dramatic increases in the scale of EBMs, but not without the difficulty of retooling models to take full advantage of the hardware (Dongarra, Gannon, Fox, & Kennedy, 2007). Kim, Tsou, and Feng (2015) used a HPC cluster of homogeneous computers to create the classic Schelling (1971) segregation model basing it on the geography and household data of San Diego, California. For the one million households and housing units, the EBM achieved almost a 180 times speed multiplier across 112 LPEs in the **computer cluster**. Another classic EBM, Sugarscape (Epstein & Axtell, 1996), was also parallelized in a homogeneous HPC **cluster**. Shook, Wang, and Tang (2013) scaled the Sugarscape model from sixteen LPEs to 2,048 LPEs for a speed multiplier of almost 1,000 times. They noted that some executions had to be

considered outliers and removed from the results dataset. This was attributed to noise caused by sharing the HPC with other users, an effect of the social scientist not having full control over the computing resource.

Heterogeneous HPCs can have a worldwide geographic distribution of dissimilar computing resources. Yang, Ono, Kurahashi, Jiang, and Terano (2015) showed this dissimilarity caused additional challenges when measuring model speed multiplier with any results necessarily normalized to the slowest computer in the grid. The slowest computer can be different for each model execution. EBM experiments showed a near linear increase in speed as the number of computers in the grid increased from one to one hundred. The D-MASON (Cordasco, et al., 2012) framework is a distributed, parallel version of the single computer EBM framework MASON. In tests contrasting the two frameworks, some models that could not be executed using MASON because of hardware limitations could be solved using D-MASON (Cordasco, et al., 2013).

These five categories shown in Table 1 of available hardware demonstrate a fragmented approach to parallelization of EBMs. Each of the hardware architectures have strengths and weaknesses that indirectly affect the methods that a social scientist must employ to reproduce, but not replicate (Drummond, 2009), an existing model or create a new one. On the positive side, this fragmentation is to be expected and commended. Social scientists are exploring all routes toward a collective goal of creating parallel models. On the negative side, the social scientist can waste time and money by choosing a parallelization strategy that does not well fit the model they are trying to create. For

instance, an EBM that requires extensive communication between agents might not fare well on heterogeneous architecture with systems distributed around the world.

3. GENERAL METHODOLOGY

3.1. Measurable Results

Experiments with different hardware architectures are detailed in Chapter 4. One measurable result from each experiment is the model **speed multiplier**. Speed multiplier compares the time it takes a model to complete using a single **LPE** to completion times using multiple LPEs. The number of agents is also changed to show the impact of scaling. Agent scaling also affects speed multiplier. This measure is useful for determining the significance of the hardware architecture in solving the social scientist's computational resource limitation. Readers are reminded of the layers shown in Figure 1 that exist above the hardware. These other layers may also have an effect on speed and could limit speed comparisons between hardware architectures.

3.2. Experiment Structure

Each hardware experiment in Chapter 4 is first introduced and followed by a brief hardware description and reasons why it was selected. The operating environment and programming language is then depicted. The approach used to implement the **EBM** for the hardware is described. Finally, the experiment, results, and any recommendations are provided.

3.3. Programming Language Selection

Several factors were considered regarding the selection of the programming language used for each hardware architecture. **EBM** frameworks were rejected because there is no assurance that the framework developers had taken full advantage of the underlying hardware. These developers must make design decisions regarding the anticipated general use of the framework that could be inconsistent with the needs of the social scientist's simulation. Programming languages are closer to the hardware and makes it easier to see how the hardware created or solved EBM challenges. The programming language was selected based on general availability, the EBM that was to be implemented, and the hardware architecture. This process aided in answering research question three, "Is it worth it?"

3.4. The Zero-Intelligence Traders Entity-Based Model in Parallel

The Zero-Intelligence Traders (**ZIT**) model (Gode & Sunder, 1993) is used for parallel **EBM** testing. It simulates a market with buyers and sellers who submit random offers and bid for a single type of a notional item. These traders have no means to observe other traders and have no memory of the past market. They also use no learning methodologies. Thus, they are deemed to have no "intelligence." Traders have pre-set randomly selected limits and will not conduct a trade that violates those limits. Those who have yet to conduct a trade are randomly matched. If a seller's minimum price and buyer's maximum price limits can be met, a randomly selected value between those limits is selected as a negotiated price. The trades are constrained so that each results in a profit. Despite the random decision-making, the ZIT agents are able to achieve market

efficiency. This led to the conclusion by Gode and Sunder (1993) that it is market institutions, and not traders, that have the greatest effect.

The simulation can be implemented sequentially by iterating through the list of buyers and sellers until a predetermined number of trades *t* have occurred. Parallelizing the model entails dividing the buyers and sellers into *n* groups (where *n* is the number of parallel processes), passing access to the data for each group to *n* computational units, executing *t*/*n* trades at each computational unit, and aggregating and summarizing the results as shown in Table 2. There is no need for a groups of buyers and sellers to communicate with buyers and sellers in other groups during the trade decisions, thus each group is independent.

Table 2 - Pseudo Code Parallelization of the ZIT Model

INSTANTIATE and INITIALIZE BUYER, SELLER, DATA and THREAD objects; Assign sub-populations of BUYERS and SELLERS to THREADS; FORK all THREADS; FOR each THREAD, REPEAT: - Randomly activate 1 BUYER agent + 1 SELLER agent: - - BUYER proposes a BID price; - - SELLER proposes an ASK price; - - IF (BID > ASK) THEN - - - Pick EXECUTION price between BID and ASK; - - - INCREMENT BUYER holdings; - - - DECREMENT SELLER holdings; - - - Collect DATA on the trade; - INCREMENT the attempted number of trades; - END when maximum trade attempts exceeded; JOIN all THREADS; Collect final DATA;
(Source: McCabe, et al., 2016)

4. APPLIED PARALLELISM

4.1. Multi-Core Central Processing Unit

Parallelization of the **ZIT** model on a multi-core **CPU** in the C programming language is accomplished by using a fork/join method. ZIT is considered an "embarrassingly parallel" problem because each trade transaction decision is independent of any other decision. This lack of communication makes it easy to split the possible transactions as blocks among different threads. These threads are assigned (forked) to the individual CPUs where the results of each block are determined before aggregating the results (joined) on the initial thread. Scaling is accomplished by increasing the number of assigned threads until some upper limit is reached (McCabe, et al., 2016).

The multi-core version of the ZIT **EBM** is attributed to source code made available by Robert Axtell (2011). This code was the basis for all experiments in this chapter.

4.2. Graphic Processing Unit

4.2.1. Introduction

CPUs measure their cores in the ones or tens, while **GPU**s, as depicted in Figure 2, do so in the thousands. The CPU is designed for complex sequential operations against small chunks of data. GPUs can perform simple parallel

operations against comparatively large amounts of data (NVIDIA Corporation, 2017d). GPUs are dependent on the presence of a GPU. The CPU has no such dependency on the GPU. The simplicity of the GPU aligns well with the idea of the **EBM** that depends on a few, simple rules for its agents coupled with a very large number of agents or geographic space that must be processed during each step of a model.

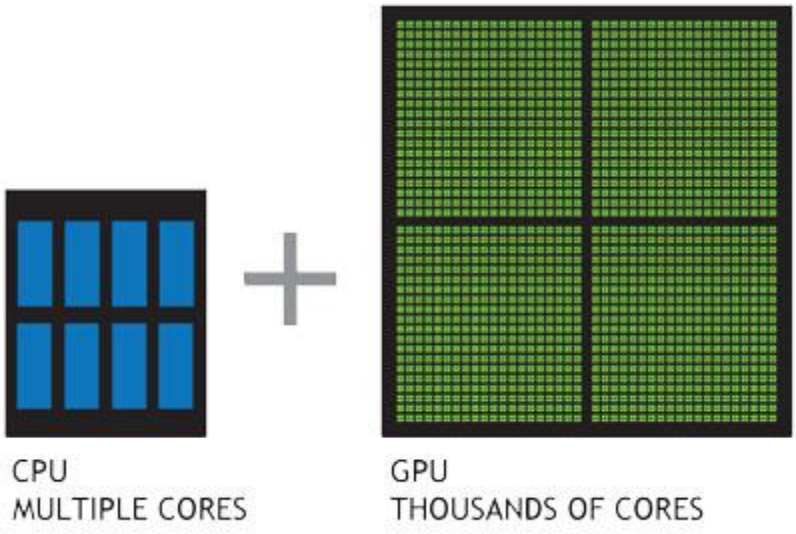

Figure 2 - Comparison of CPU to GPU Cores
(Source: NVIDIA Corporation, 2017d)

There is more than one manufacturer of **GPGPU**s with NVIDIA (http://www.nvidia.com) and AMD (https://www.amd.com) as the most widely known. Terminology specific to the NVIDIA GPGPU is used in this thesis. The concepts, however, are the same in other manufacturer's implementations.

4.2.2. Hardware

The hardware platform selected for this experiment is the NVIDIA Jetson Tegra X1 (TX1) Developer Kit. Relevant specifications for the **CPU** support and the **GPU** are in Table 3. This platform was intended as a complete environment for software and hardware developers, particularly those needing high performance computations (NVIDIA Corporation, 2017b). The GPU is embedded in its own board that plugs into the main board.

Table 3 - GPU Hardware

• Quad-core ARM® Cortex®-A57 MPCore Processor • 4 GB LPDDR4 Memory • 16 GB eMMC 5.1 Flash Storage • 10/100/1000BASE-T Ethernet • 5 MP Fixed Focus MIPI CSI Camera • NVIDIA Maxwell™ GPU with 256 NVIDIA® CUDA® Cores
(Source: NVIDIA Corporation, 2017b)

4.2.3. Justification

This hardware provided a clean environment bereft of additional applications that could affect available system resources. Access was accomplished via a command line interface, eliminating the overhead of a graphical user interface (GUI).

4.2.4. Operating System and Environment

The TX1 was pre-installed with the Linux operating system, Ubuntu distribution release 14.04 LTS. JetPack 2.2 for L4T was used to load the additional software necessary for interacting with the **GPU**.

4.2.5. Programming Language

Programming for the NVIDIA **GPU** hardware was accomplished through the Compute Unified Device Architecture (**CUDA)** 7.0 computing platform. The CUDA environment provides a means to transfer data between the **CPU** main board (host) and the GPU. It also manages the transfer and execution of programs from the host to the GPU. CUDA operates as either an extension of languages such as C, C++, and FORTRAN or as an Application Programming Interface (**API**) for other languages such as Python (NVIDIA Corporation, 2017a; NVIDIA Corporation, 2017c).

For this experiment, the host language used is C under **GNU**'s Not Unix (GNU) Compiler Collection (**GCC**) (https://gcc.gnu.org/) version 4.8.4. The acronym GNU is a recursive acronym. The code was compiled using NVIDIA's CUDA Compiler (**NVCC**).

4.2.6. Approach

Programming a **GPU** requires a different coding approach than that for a **CPU** only. The GPU is reliant on the CPU no matter which programming language is chosen. Data must be prepared in the CPU prior to invoking the GPU.

Once prepared, the data is moved to the GPU along with a set of programming instructions that will execute against the data. Upon completion, the data results are moved back to the CPU where additional processing can occur. This communication with the **GPU** is accomplished using **CUDA** kernels. These sections of code use a triple angle bracket syntax <<< >>> that notifies the CUDA compiler it is a GPU function call.

The triple angle bracket syntax plays another important role. Contained within the brackets is information informing the GPU of the number of CUDA cores to allocate to the task. Typically, two values representing blocks and threads are provided while a third value for grids is optional. Figure 3 shows the relationships between these resources. Unlike the CPU where the thread is a software process, the GPU thread is a single CUDA core, a hardware process. CUDA GPUs are best executed in blocks of 32 threads referred to as warps. This is considered a best practice methodology (NVIDIA Corporation, 2012). Consequently, thread blocks should always be in multiples of 32. The block value indicates the number of thread blocks to execute, each of which will execute a number of threads that have been allocated. Each GPU model has an upper limit for the number of threads that can execute within a single thread block. It is most efficient to use the maximum number of threads available within a thread block while never exceeding the upper limit, 1,024 for this GPU. Violating the upper limit will cause the code to fail (NVIDIA Corporation, 2017f). A CUDA command is available to discover properties unique to each GPU model.

One final note is the generation of pseudo random numbers. **EBM**s often use these numbers to explore a model's decision space during multiple runs. The GPU cannot make host system calls to generate pseudo random numbers. Consequently, a separate CUDA pseudo random number generator must be used on the GPU. This can result in two different pseudo random number generators within the same model.

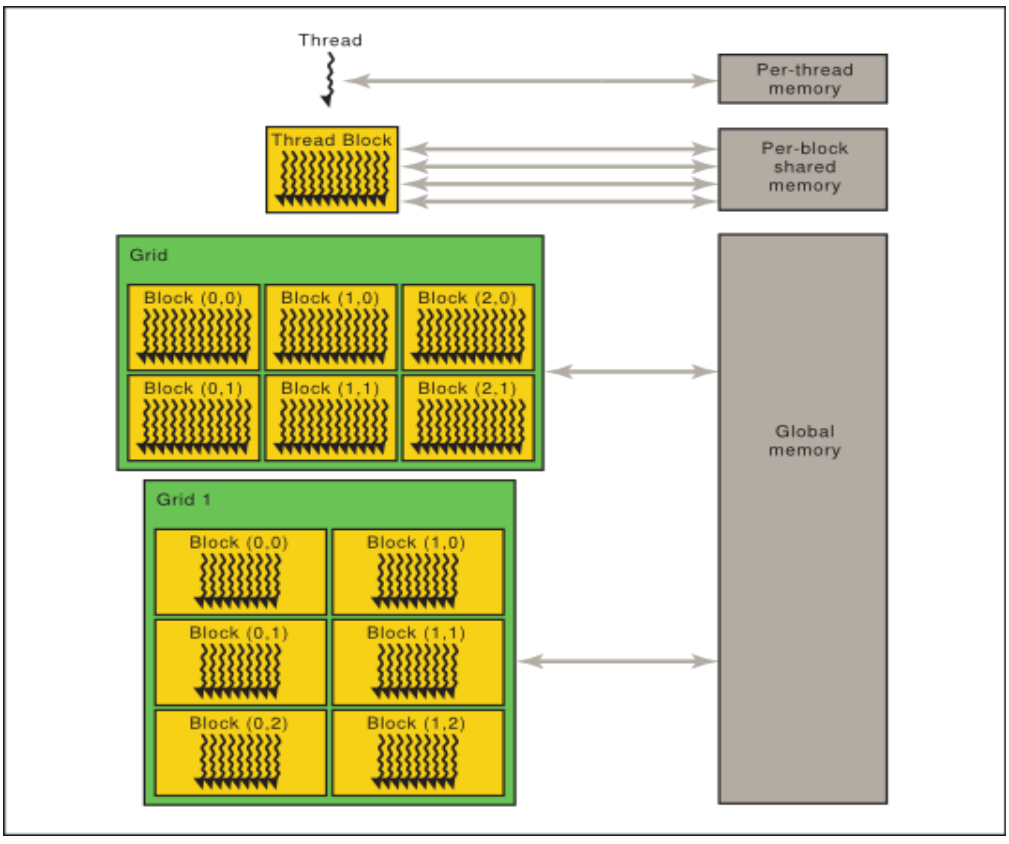

Figure 3 - GPGPU Threads, Blocks, Grids, and Memory
(Source: NVIDIA Corporation, 2017e)

4.2.7. Experiment and Results

The **GPU** code was tested via a parameter sweep varying both the number of agents and the number of desired threads. The number of agents was tested from 10^3 through 10^6 by factor of ten increments against the number of threads from 10^0 to 10^6, also by factor of 10 increments. The number of trades was always 10^2 greater than the number of agents and the maximum buyer and seller values were sent to 30. Table 4 shows the mean and standard deviation in milliseconds of 30 runs for each valid combination of parameters. Six parameter combinations were invalid, as the number of agents cannot exceed the number of threads. The measured times are only for the GPU kernel that calculated trades. Sequential **CPU** time and other preparations were not measured.

Table 4 - GPGPU ZIT Results

	Mean in Milliseconds				Standard Deviation in Milliseconds			
Threads	**10^6 Agents**	**10^5 Agents**	**10^4 Agents**	**10^3 Agents**	**10^6 Agents**	**10^5 Agents**	**10^4 Agents**	**10^3 Agents**
10^0	158,001	50,929	5,916	306	21,678	13,643	2,432	53
10^1	56,655	9,851	756	45	10,106	2,598	169	19
10^2	33,566	2,333	184	26	21,664	374	26	12
10^3	17,353	1,530	142	20	5,235	129	24	8
10^4	9,430	789	66	---	3,483	35	14	---
10^5	8,588	702	---	---	3,378	2	---	---
10^6	4,806	---	---	---	2,461	---	---	---

The model used the best practice methodology of running threads in warp sized blocks as described in Section 4.2.6. Consequently, the actual number of threads used was sometimes reduced to the nearest warp multiple that did not exceed the desired number of threads. Thus 10^2, 10^3, and 10^4 desired threads were reduced to 96, 992, and 9,984 respectively. This best practice was violated for desired threads of 10^0 and 10^1 as the result would have been zero threads. Instead, the desired number of threads was used. The model ensured the number of threads per block did not exceed the upper limit by adjusting the number of thread blocks to accommodate the additional threads. In doing so, the model also ensured the number of threads per block was the same for all thread blocks. While selecting desired threads in factors of 10 made near maximum use of available threads per block, intermediate values would use smaller values when favoring an even distribution of threads across all thread blocks.

The **speed multiplier**, relative to a single thread, for each number of agents from Table 4 is shown in Figure 4. Adding additional threads monotonically decreased computation time resulting in a relative speed increase. Allocating a single thread, and therefore a single **CUDA** core, for each agent's computation was consistently the best strategy.

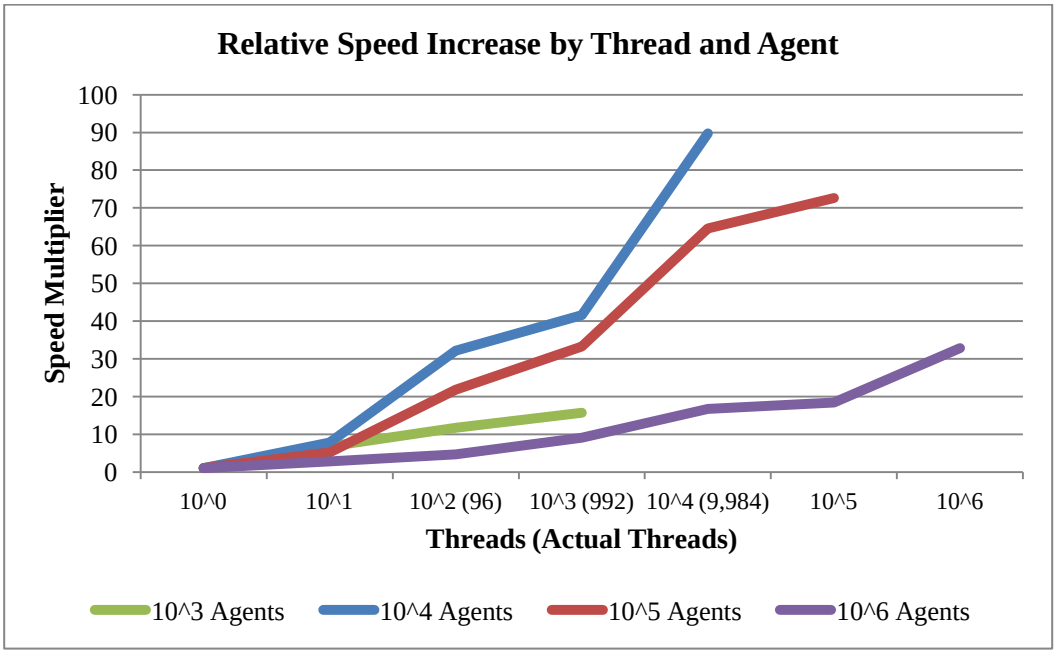

Figure 4 - Speed Multiplier Results Using a GPGPU

4.2.8. Recommendation

Coding for a **GPGPU** requires a willingness on the part of the social scientist to acquire a deep understanding of the hardware. Future **CUDA** platforms will no doubt continue to add useful layers between the hardware and the programming language, possibly at the expense of additional execution time. For example, an addition to the CUDA 6.0 platform, Unified Memory, eliminated the need to explicitly declare separate GPU and **CPU** memory structures and explicitly move data between them. However, Unified Memory takes slightly longer to execute (NVIDIA Corporation, 2013). Training for CUDA platforms is readily available at NVIDIA, free online courses, and academic institutions. The

benefit to the social scientist is a capability to conduct large-scale experiments with commonly available and low cost equipment. (The TX1 used in this experiment was acquired from NVIDIA for $300 at a 50% education discount.) The CUDA platform is capable of using multiple GPU boards attached to the same CPU host, adding additional computing power.

4.3. Application Specific Integrated Circuit

4.3.1. Introduction

Application Specific Integrated Circuits (**ASIC**) are purpose designed to implement some function in hardware, often as an accelerator for some process. Examples of this include Digital Signal Processing (e.g. McCanny, Ridge, Hu, & Hunter, 1997), Software Defined Networks (e.g. Zaho, Li, Han, Sun, & Huang, 2014), and Neural Networks (e.g. Nurvitadhi, et al., 2016). Similar to the **GPU**, the ASIC is dependent on a **CPU** host. Only some ASICs are useful for implementing an **EBM**. The Adapteva's Epiphany III (Adapteva, 2013) is such an ASIC as it is designed for general parallel processing.

4.3.2. Hardware

The Epiphany III has sixteen independently operating **RISC** computing nodes (**cores**) on an internal mesh network. This **network on chip** (**NoC**) is capable of simultaneous read, on **chip** write, and off chip write operations as shown in Figure 5. Each **core** has its own 32 KB memory plus access to 512 MB memory shared by all nodes. The design of this **ASIC** supports **Single**

Instruction Multiple Data (**SIMD**), **Single Program Multiple Data** (**SPMD**), and **Multiple Instruction Multiple Data** (**MIMD**) among others (Adapteva, 2013).

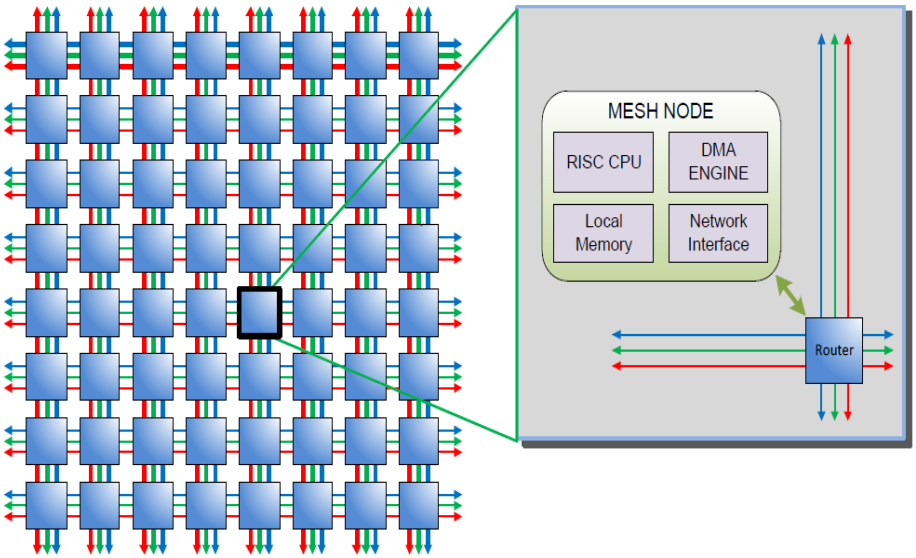

Figure 5 - The Epiphany Architecture (64 cores shown) (Source: Adapteva, 2013)

The Epiphany is implemented as a co-processor to a **CPU** (itself a RISC ARM 9 processor) on a single Adapteva Parallella board. While this negates compatibility concerns between the CPU host and the ASIC, it limits hardware scalability of easily adding another board to a CPU host. The Parallella has 1 GB of memory, half of which is the shared memory for the Epiphany.

4.3.3. Justification

The single board Parallella offers an integrated **CPU**/**ASIC** environment designed for experimenting with **SPMD**. All computations can occur on the Epiphany while the CPU is in a support role. Access was accomplished via a command line interface eliminating GUI overhead concerns.

4.3.4. Operating System and Environment

The Parallella provides an Ubuntu distribution of the Linux operating system Release 14.04. Interactions between the **CPU** host and the **ASIC** are implemented through libraries and applications that do not interfere with the host operating system.

4.3.5. Programming Language

The **ZIT** model is implemented in Epiphany BASIC (eBASIC version 0.1) (Brown, 2015). eBASIC is a dialect of the BASIC computer programming language, a procedural language developed in the mid-1960s (Ralston & Meek, 1976). Additional commands specific to the Parallella operating environment and parallel processing were added by Brown to this subset of the BASIC language.

4.3.6. Approach

An **SPMD** approach is used to parallelize **ZIT**. The **CPU** host distributes identical copies of the eBASIC ZIT model to each of the sixteen **cores** on the Epiphany III (**SPMD**). An area of shared memory, accessible to all cores is initialized with seller and buyer data. Each distributed program can determine its

unique core identification number and uses this, along with the maximum number of cores, to determine which section of shared data is unique to the core. Each core executes its code independently until all trades are complete. One core, selected as a master, has the additional task of collecting results from the other cores and calculating the overall results. Used in this way, the **ASIC** cores are functionally identical to software threads.

The SPMD approach minimizes changes to the original model. Porting the ZIT model from the C programming language to eBASIC actually simplified the model, as there was no reason to create a large array of buyers and sellers that was divided and processed via a fork/join process. Instead, each eBASIC copy used its own data set where the array size was simply limited to '1/*maximum-number-of-cores*'. Because of limited memory, however, it was necessary for the data to be placed in the 515 KB shared memory rather than local core memory. The program itself, at 1,724 bytes, easily fit on each core.

4.3.7. Experiment and Results

The parameter sweep for the **ASIC** code included the number of agents and **cores**. The number of cores was incremented from 1 to 16 in steps of one while the number of agents was tested from 10^3 through 10^5 by factor of ten increments. A test of 10^6 agents could not be performed given memory limitations. The number of trades was set to 10^2 greater than the number of agents and maximum buyer and seller values was 30. Table 5 shows the results of

30 runs for each parameter combination. Moving data between shared in core memory, although transparent to the program, negatively impacted performance. Differences in relative **speed multiplier** for different number of cores are shown in Figure 6. The maximum speed multiplier for this ASIC is an order of magnitude smaller than for the **GPGPU**. Despite an increase in the number of agents, the speed multiplier remains nearly identical for all cores with a value of 8.65-8.69 for 16 cores.

Table 5 - ASIC ZIT Results

	Mean in Milliseconds			Standard Deviation in Milliseconds		
Cores	**10^5 Agents**	**10^4 Agents**	**10^3 Agents**	**10^5 Agents**	**10^4 Agents**	**10^3 Agents**
1	986,110	98,693	9,874	437	81	7
2	494,684	49,554	4,962	390	36	5
3	332,037	33,231	3,328	265	28	3
4	251,713	25,184	2,525	188	20	2
5	206,530	20,660	2,070	140	15	3
6	176,471	17,658	1,767	105	13	2
7	155,705	15,580	1,561	87	7	2
8	140,431	14,056	1,408	89	14	1
9	130,789	13,091	1,316	84	5	1
10	123,955	12,404	1,246	157	12	1
11	119,898	11,995	1,206	117	18	3
12	117,324	11,741	1,178	75	11	2
13	116,556	11,668	1,172	107	13	1
14	116,121	11,618	1,166	71	9	1
15	115,007	11,509	1,161	46	5	15
16	113,511	11,359	1,142	115	17	5

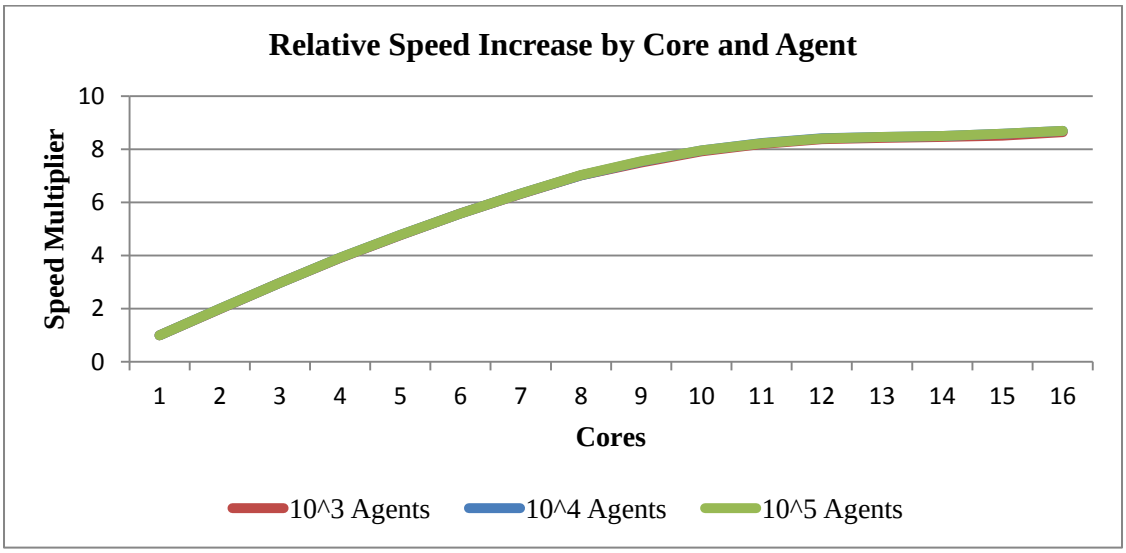

Figure 6 - Speed Multiplier Results Using an ASIC

4.3.8. Recommendation

ASIC coding will vary depending upon the type of accelerator. However, any general parallel processing accelerator that permits **SPMD** or **MPMD** has potential to assist the social scientist with **EBM** development. In particular, an ASIC that reliably scales across agent size would be a stable predictor for system design. The low cost of the Parallella hardware ($100) is ideal for creating a computing cluster provided the problem's data can be broken into small chunks. Future research could include daisy chaining multiple boards together extending the number of available cores. Additionally, each Parallella host **CPU** and its operating system can communicate via 1 GB Ethernet. This creates an opportunity for real-time model monitoring.

4.4. High Performance Computing Cluster

4.4.1. Introduction

A High Performance Computing (**HPC**) cluster provides a scalable environment for computations on large amounts of data that is distributed across multiple data nodes.

4.4.2. Hardware

A set of homogeneous data nodes store the distributed data, each of which also provides the necessary computing resources. These data nodes, along with some additional computers for resource management, are commodity systems. Commodity systems are general-purpose hardware typically appropriate for servers. The **HPC** cluster used for this experiment consists of 11 data nodes. Each node has 24 **LPE**s (hyper-threaded from 12 physical **CPU**s in a single **IC**) and 64 GB of memory.

4.4.3. Justification

Scalability is the primary justification for experimenting with this hardware. Several commercial companies offer the ability to rent a **HPC** of almost any size for short time periods. Thus, there is the potential for the social scientist to engage in very large scale **EBM** simulations. The hardware used in this experiment was a low use development system available at no cost. Access was via command line eliminating any **GUI** overhead.

4.4.4. Operating System and Environment

The Cloudera 5.x platform (http://www.cloudera.com/) running on top of Linux was used as the computing environment. Cloudera uses a **Hadoop** distributed file system. This system emphasizes keeping data at rest since data movement takes time. This is accomplished by breaking the data into chunks, replicating each chunk so there are three identical copies, and distributing all chunks across the data nodes. Once placed, the data does not move, instead computing tasks are sent to the data. Cloudera uses the resource manager **YARN** to distribute work across the cluster and then collect the results.

4.4.5. Programming Language

The **ZIT** model was implemented in PySpark 1.6. Spark (Apache Software Foundation, 2016) is a general processing engine for use in **HPC** environments. PySpark is a Spark Python **API** that extends the Python programming language to the Spark programming model. Spark works with **YARN** to distribute multiple tasks across the cluster as shown in Figure 7.

4.4.6. Approach

Parallelization of the **ZIT** model using this hardware requires a modified approach. The underlying system communicates and stores data in a fundamentally different way than that of a single computing system. Since the data is immutable, transactions must take place using a series of map and filter operations. A map operation applies some function to each element in a list,

returning a list of results. A filter operation applies a set of criteria to each element in a list and returns a subset of the original list. The result is a functional programming approach to the problem rather than the procedural approach used by some other languages.

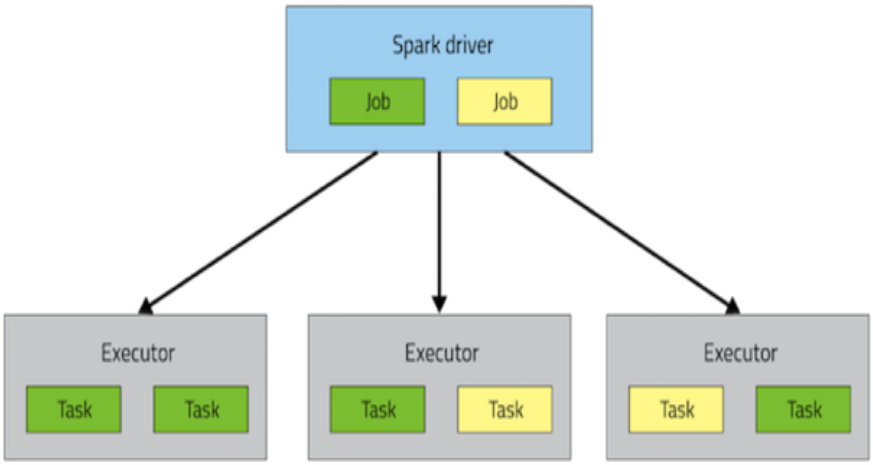

Figure 7 - How Spark Manages Work
(Source: Cloudera, 2015a)

Unlike hardware using just physical cores or software threads, this **HPC** cluster necessitated a two dimensional approach to resource allocation. The concept of threads still exists as **cores**, but increasing the number of cores alone does not decrease execution time in a generally monotonic fashion. Instead, each data node also has executors that are responsible for implementing tasks on each core. Therefore, computational resources are assigned by stating the number of

executors (1-*n*) and cores (1-*c*) (where *n* >= 1 and *c* >=1 and *c* <= the number of **CPU**s on the data node) to be used on the problem.

4.4.7. Experiment and Results

Given the two dimensions of resources, the first experiment was to discover the most efficient combination of **cores** and executors. It became clear there was no standard answer as a number of information sources, including Cloudera, all proposed different solutions. In a two part blog posting (Cloudera, 2015a; Cloudera, 2015b) offered several operating environment parameter adjustments for finding the best solution. Apache Spark went further, including adjustments accounting for the physical relationship between the compute nodes and the data locality (Apache Software Foundation, 2017). Garbage collection could also make a difference (Wang & Huang, 2015). In each case, no single solution was offered.

Using the available **HPC**, the **ZIT** model is run 30 times each using a parameter sweep from 1-44 executors and 1-24 **cores** with 10^4 agents and 10^6 trades. The maximum prices for the agents were set to 30. The results in Figure 8 show the average execution time (in seconds) in two dimensions of the effect of varying these two parameters. The maximum number of seconds was artificially capped at 43.8 seconds (the average across all results plus 3 standard deviations) to better visualize the differences between regions. The horizontal line shows the hyper-thread boundary of the twelve physical **CPU**s and shows the effect of

hyper-threading under some conditions. The number of data nodes also created a boundary condition at two times the number of data nodes (the right vertical line). Some combinations were always sub-optimal. For example, using a single executor always resulted in a poor time no matter how many cores were used. Using three or fewer cores was also a poor choice if the number of executors was not more than the number of data nodes. There were also unexplained slow areas, such as when six executors were used.

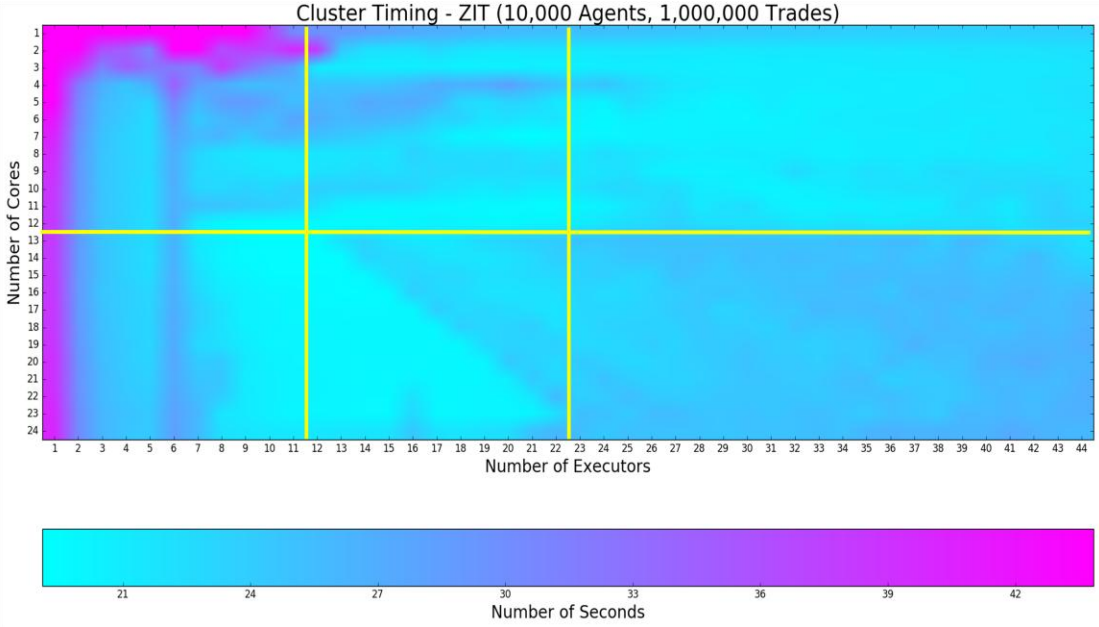

Figure 8 - ZIT HPC Optimization Parameter Sweep Results

Optimal areas were discovered. Figure 9 shows the fastest execution times took place when using 10 to 12 executors coupled with 12 to 15 cores. This is in contrast to applying a maximum resource of 44 executors and 24 cores. That fewer computational resources results in a faster execution time at first appears counterintuitive. However, this may be the result of increased communication interactions between the data nodes at the operating environment level.

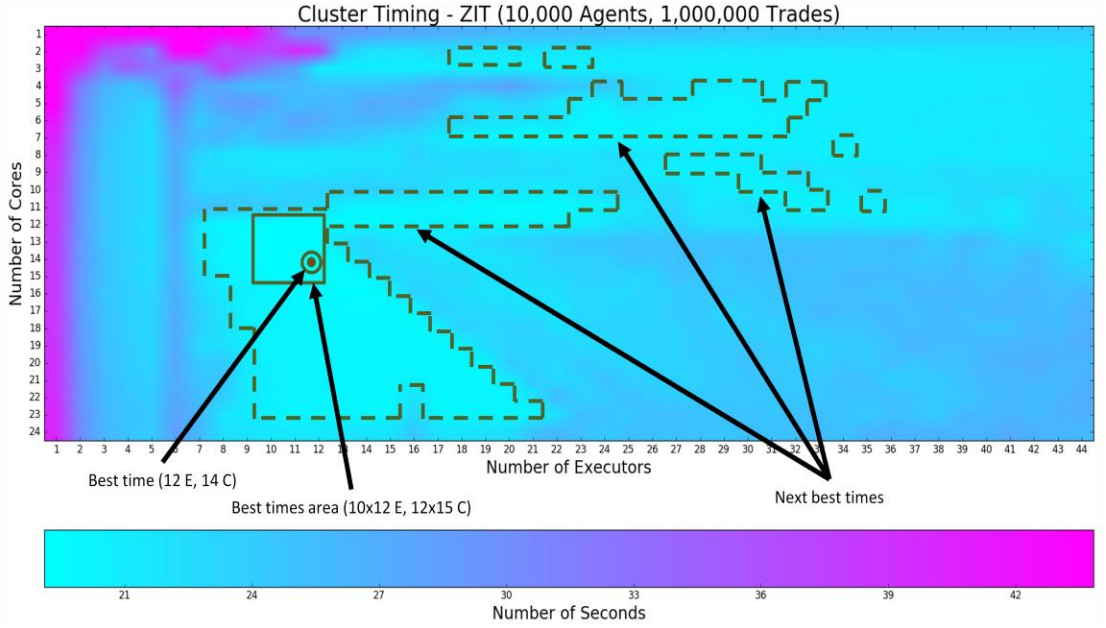

Figure 9 - ZIT HPC Optimization Analysis

The ZIT model itself required no inter-agent communication during execution. Each data node operated independently until local results were

reported back to the top-level job. Increasing the number of executors beyond the best time area, usually led to slower model execution times. This was particularly noticeable when the number of cores was also increased to the point that hyper-threading was engaged. Decreasing execution time may be best accomplished by increasing the number of data nodes. Doing so will most likely change the executor/core boundaries that delineate the area of best execution. The social scientist will need to gather information regarding their hardware environment before committing to a large run of their **EBM**.

Table 6 - ZIT Time to Complete on HPC Cluster

Agents	Mean in Milliseconds	Standard Deviation in Milliseconds
10^3	17,688	813
10^4	19,068	765
10^5	34,427	734
10^6	180,999	3,971
10^7	1,714,556	17,418

Using the discovered best combination for this problem, the second part of the experiment is performed. Selecting 12 executors and 14 cores, a parameter sweep of 10^3 through 10^7 agents is conducted. The number of trades is set to be always 10^2 greater than the number of agents. Maximum agent trade values are set to 30. For each set of agents, 30 runs are conducted with the results shown in Table 6 and graphically in Figure 10. Previous speed multiplier calculations were

predicated on the single dimension of a **LPE**. The addition of executors in a second dimension precludes a simple speed multiplier calculation. Consequently, the comparison of agent scaling in Table 6 and Figure 10 is shown in raw execution time.

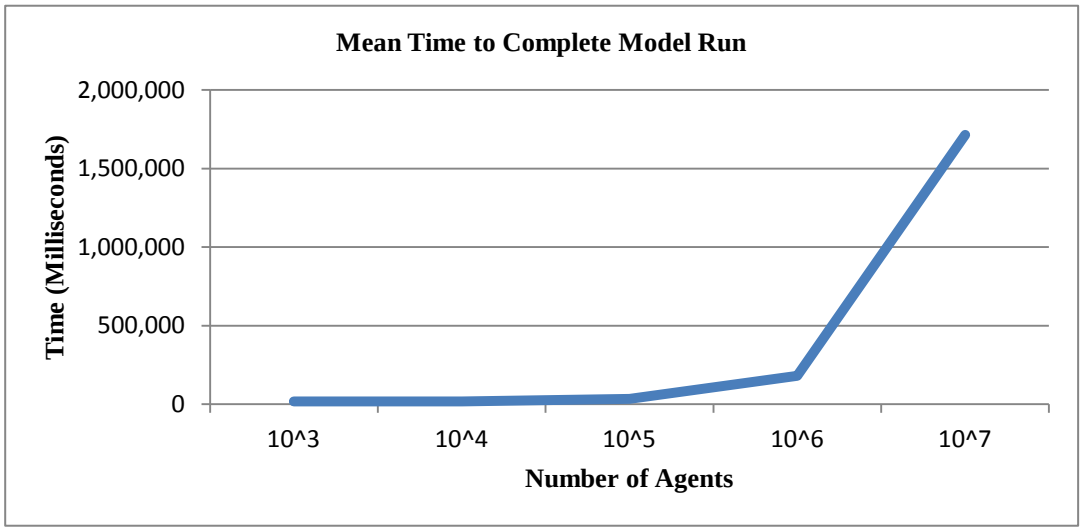

Figure 10 - HPC ZIT Speedup Results

4.4.8. Recommendation

Using an **HPC** cluster could be an easier route for the social scientist as they are separated from direct interaction with the hardware in most cases. Various providers such as Amazon, Google, and Microsoft offer free training for their commercial offerings and rent time on a HPC that can be sized to the need of

the social scientists. Best use would be to create a small cluster used to learn the operating environment and test the viability of the model. Then scale the number of data nodes testing the speed of completion and the number of **cores** and executors. Finally, the full-scale model could be run with the maximum agents desired. For those social scientists desiring direct interaction with the hardware, it should be noted that all of the software, Linux, **Hadoop**, **YARN**, and Spark are available as open source at no cost. The software may be used on hardware as simple as the Raspberry Pi.

5. MODEL VERIFICATION AND VALIDATION

5.1. Verification

The **ZIT** model is a simple model, as previously illustrated in Table 2. The pre-existing code used for **CPU** parallelization was used as a source for porting to the new hardware. Each model's code was verified as correct by first correcting any implementation syntax errors in the programming language. Once correct, the outcomes of any functions were examined by using test data with known outcomes. Finally, the assembled code was run using a small number of agents and trades to ensure end-to-end completeness.

5.2. Validation

The purpose of validation is to, "show that the model actually works in a similar fashion to the real world" (Wilensky & Rand, 2015, p. 335). The **ZIT** model, however, is a theoretical model with no direct connection to real world trading. A trading model can be considered structurally valid (Cioffi-Revilla, 2014, pp. 297-299) if it is internally consistent with the manner in which real traders operate, no matter how simply. For a trade to occur there must be at least one willing buyer and one willing seller. The ZIT model does follow this simple rule. The model also provides the mean and standard deviation of the overall trade results for each run. These values were examined and, despite the number of agents and trades changing, were found to be internally consistent.

6. SUMMERIZED RESULTS AND CONCLUSION

This concluding chapter restates key research findings in Section 6.1. Suggestions and opportunities for extending this thesis are found in Section 6.2, and some concluding thoughts are in Section 6.3. Table 7 recaps the research questions and findings.

6.1. Overview of Research Findings

Simple **SPSD EBM** frameworks such as NetLogo can quickly run out of computational resources as a model increases in size. Bypassing frameworks and directly using the underlying programming languages allows more flexibility, but are still constrained unless a language is fully integrated with the hardware capabilities. Three research questions were explored. 1) Does the underlying hardware play a role in the social scientist's capability to create large-scale models? 2) If so, does the hardware change the approach and skills needed for modeling? 3) Is it worth the effort?

A well-coded model in any programming language is still restricted by the computational resources made available by the hardware. A fast **CPU** operating sequentially is limited by the time it takes to execute a single instruction and the size of memory that contains the model. Dividing the problem across multiple computational resources decreases the time to complete a model run. To explore its problem space, an EBM may need to be run thousands of times with different parameters. Figure 11

provides a comparison of the raw execution time results from all three experiments. For each, the intersection of agents and threads shows a circle whose area represents the mean execution time in milliseconds. (An approximation of threads for the **HPC** model was calculated as 168, 12 executors times 14 **cores**.) The **GPGPU** achieved the fastest execution time, but the HPC was able to handle a greater number of agents. The **ASIC** was the slowest hardware, but provided uniform scaling across all agents as shown in Figure 6. The ASIC was also the easiest to implement using an **SPMD** approach. All experiments in this thesis demonstrated a decrease in execution time, and increase in **speed multiplier**, by utilizing parallel capable hardware. Decreasing run time permits larger models to be executed with the same period. The first research question is answered: **The correct hardware can increase the social scientist's capability to create large-scale models by allowing for the possibility of parallelism.**

The **ZIT** model was chosen because of its simplicity. This embarrassingly parallel model only requires agent communication at the conclusion of program execution. Despite its simplicity, implementation in each experiment required a different approach. The first step was to obtain a deeper knowledge of the hardware than is required for sequential computing. Surprisingly the **HPC** cluster required the least amount of hardware knowledge. As a collection of commodity computers, it was only necessary to know the number of data nodes, **LPE**s per data node, and the amount of memory available at each data node. Discovering the appropriate amount of computational resources for the problem, however, was more complicated as it required a two-step process and a two dimensional solution.

The ASIC as a coprocessor presented itself ready for SPMD. Therefore, understanding the access and use of a single LPE was identical to understanding multiple LPEs. Any hardware offering SPMD should be sought after by the social scientist.

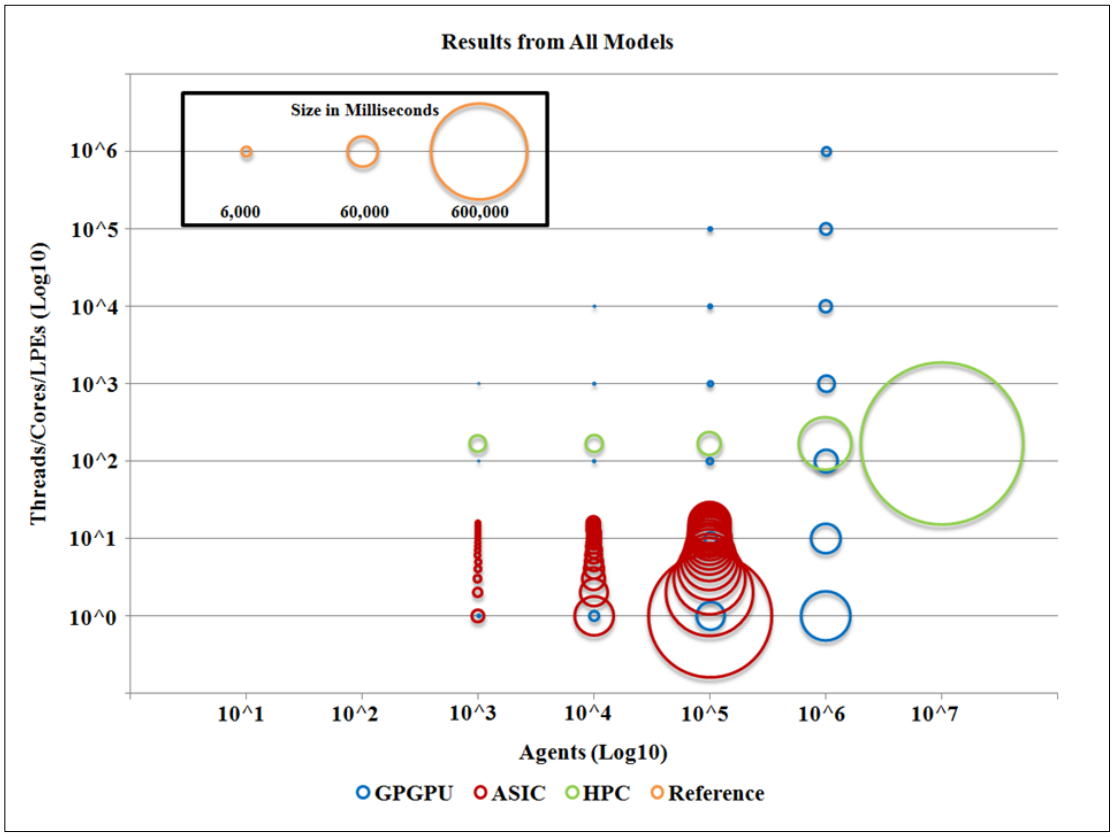

Figure 11 - Comparison of Three Model Results

The GPGPU required the most effort in understanding the hardware. Originally designed to speed the display of images on monitors, this hardware was quickly diverted into a role as a computational resource. Its beginnings as a **GPU** tie it to smaller memory

than is typically available on the host and to simpler computational LPEs. Careful attention to memory management becomes critical, quirks such as warp size may modify the model, and the simple computational **core** will not support some functions normally available to a programming language.

The second step was to understand the affect hardware had on the programming languages that in turn affected the programming approach. The ASIC hardware supporting SPMD required but a single program that could be distributed without further modification to all LPEs. The programming language eBASIC is a sub-dialect of decades old simple language to which has been added a small number of extensions specific to the Parallella, parallel processing, and concurrent processing. Despite the simplicity of the language, it was fully capable of implementing the ZIT model. The language, combined with the SPMD architecture, made implementation easy, as very few hardware specific changes were necessary. Not as easy was the GPGPU implementation that leaned heavily on the **CUDA** environment. As an extension to the C programming language, CUDA kernels mimicked C functions. Thus, it is necessary to understand how CUDA communicates with the hardware. Any difficulties were with ensuring GPGPU resources were properly allocated through the kernel interface. Fortunately several CUDA error checking functions are available that should be understood and used routinely by the social scientist. Finally, the HPC cluster required the ZIT model to be written using a functional programming approach. A data set of pre-trading agents was produced and distributed to the data nodes. Once created the data set was immutable thus could not be changed. A sequence of functions and filters were applied to the immutable

data to transform it into a new dataset with the trade results. The second research question is answered: **The hardware does change the approach and skills needed for modeling**. **The amount of change and the level of programming skills will vary between hardware types. SPMD co-processer architecture may require the least additional skills.**

In most cases, social scientists would be best advised to use an EBM framework or sequential programming. Not every model will have so many agents, such a large geographic space, or immense communication that a parallel EBM solution would be worth the extra effort. In such cases, however, a parallel EBM may be the only solution. For some hardware architectures, it might be best for the social scientist to collaborate with a computer scientist. The third research question is answered: **Is it worth the effort only when it is the only available solution.** Table 7 provides a recap.

Table 7 - Research Question Recap

Number	Research Question	Research Answer
1	Does the underlying hardware play a role in the social scientist's capability to create large-scale models?	The correct hardware can increase the social scientist's capability to create large-scale models by allowing for the possibility of parallelism.
2	If so, does the hardware change the approach and skills needed for modeling?	The hardware does change the approach and skills needed for modeling. The amount of change and the level of programming skills will vary between hardware types. A SPMD co-processer architecture may require the least additional skills.
3	Is it worth the effort?	Is it worth the effort only when it is the only available solution.

6.2. Future Research

An obvious research area is the creation of a parallel **EBM** framework for each of these hardware configurations and some work such as FLAME (http://www.flamegpu.com/) has been completed in this area. These frameworks were not explored in this thesis as the focus was on hardware. Parallel EBM frameworks would solve the need for the social scientist to acquire in-depth hardware knowledge.

Hardware specific research should include increasing the number of data nodes in **HPC** clusters, using multiple **GPGPU** boards on a single host, HPC clusters whose commodity computers include GPGPU boards, chaining **ASIC** boards to increase the number of **LPE**s, and exploring HPC clusters with installed ASIC coprocessors.

Finally, testing a more complex EBM in these environments may better compare the architectures. In particular, an EBM that requires agent communication during and not just at the conclusion of the model run. In this case, it may be essential to tie the locality of the agents to a computational LPE so to minimize the need to transfer information across computational boundaries.

6.3. Conclusion

Simple **EBM**s that fit within the computational limitations of a single computer executing sequential tasks may no longer be adequate for future EBM questions. The era of big data and data science has embraced the concept of analysis of populations rather than just samples. There is no reason to expect EBMs will be any different with the wide

array of available computing power. This thesis will hopefully be a benefit to the social scientist who is faced with a large-scale model and limited time.

APPENDIX A - LEXICON

As with any specialization, there is terminology specific to **EBM** parallelization that a social scientist must know.

Application Programming Interface (API): A means for a software component to communicate with another software component. A component may provide the **API** as a standardized means by which another component may request access to data or resources.

ASIC (Application Specific Integrated Circuit): An integrated circuit (**chip**) designed for some specific use. An example for parallel processing is a set of homogeneous **RISC** processors in a mesh grid on a single **chip**.

Cellular Automata (CA): A limited type of **EBM** that is of the simplest types of social simulation models. There are no agents; instead, each region in some environment reacts to its surroundings. Over time, these reactions can create the illusion of movement.

Chip: See **Integrated Circuit**.

Cluster: See **Computer Cluster**.

Computer Cluster: A collection of computers of which some or all can be focused on the same task. Resource allocation within the cluster is left to cluster management software, thus the cluster can be viewed as a single system.

Concurrent Computing: The ability to simultaneously execute several tasks that may be unrelated. Similar, but distinct from **parallel computing**.

Core: See **Logical Processing Element**.

CPU (Central Processing Unit): An electronic circuit, or **chip**, which performs low-level program instructions. Multiple CPUs on a single **chip** are referred to as **cores**. A single chip with multiple cores can be called a **socket**.

EBM Framework: A software application purposely designed to facilitate the execution of an Entity-Based Model.

GPGPU (General purpose computing on graphics processing units: The use of a **GPU** to perform non-graphic computational tasks normally performed by a **CPU**. **EBM frameworks** that use a **GPU** for computing fall under this definition.

GPU (Graphics Processing Unit): A system of parallel configurations that can efficiently process blocks of data simultaneously. Originally designed to speed up visual displays, they have been adapted to scientific computing. (See **SIMD**.)

Hadoop: Apache Hadoop is open source software that provides an environment for distributing large data sets across a **computer cluster**. Also, see **YARN**.

High performance computing (HPC): An increase in computation power achieved by aggregating multiple computing resources and focusing them on a single task.

Integrated Circuit (IC): A collection of electronic components and wires on a single flat piece of semiconductor material. An essential component of most current computing systems.

Logical Processing Element (LPE): The use of hyper-threading technology can create the illusion of multiple processors where there is but one physically. Most often, there appears to be two logical processing elements per physical element. When determining the number of available cores, software applications count these logical processing elements.

MIMD (Multiple instruction streams, multiple data streams): Multiple cores work on multiple blocks of data simultaneously.

MISD (Multiple instruction streams, single data stream): Multiple cores work on the same block of data. Useful in situations where a **core** could fail to produce correct results.

Network on Chip (NoC): One or more communication networks between **cores** on a single **integrated circuit**. The network can operate either synchronously or asynchronously.

Parallel Computing: The ability to simultaneously execute a group of tasks created by dividing an original task into many smaller identical tasks. Similar, but distinct from **concurrent computing**.

RISC (Reduced instruction set computing) coprocessor: A special purpose processor using a limited but fast set of instructions that is tightly coupled to the **CPU**.

SOC (System on a Chip): A single **integrated circuit** that encompasses all necessary components for a functioning computer.

Socket: See **CPU**.

SIMD (Single instruction stream, multiple data streams): A single set of instructions is simultaneously applied to multiple blocks of data. (See **GPU**.)

SISD (Single instruction stream, single data stream): A single set of instructions is applied to a single block of data. This is a sequential (non-parallel) process.

Speed Multiplier: A relative comparison of execution time where the base is set to the time to complete using a single **LPE** or **core**. The **Speed Multiplier** is calculated by dividing the base execution time by the execution time using multiple **LPEs.**

Speedup: See **Speed Multiplier**.

YARN (Yet Another Resource Negotiator): An operating system for large-scale distributed data processing. Used by **Hadoop**.

REFERENCES

Aaberge, T. (2004). *Analyzing the Performance of the Epiphany Processor.* Thesis, Norwegian University of Science and Technology, Trondheim, Norway. Retrieved from https://daim.idi.ntnu.no/masteroppgaver/011/11745/masteroppgave.pdf

Adapteva. (2013). *Epiphany Architecture Reference Manual.* Reference, Cambridge, MA. Retrieved from http://www.adapteva.com/docs/epiphany_arch_ref.pdf

Anderson, D. P. (2004). BOINC: A System for Public-Resource Computing and Storage. *Proceedings. Fifth IEEE/ACM International Workshop on Grid Computing, 2004* (pp. 4 - 10). Pittsburgh: IEEE.

Apache Software Foundation. (2016). Spark. Retrieved from http://spark.apache.org/

Apache Software Foundation. (2017). *Tuning Spark.* Retrieved from https://spark.apache.org/docs/latest/tuning.html

Axtell, R. L. (2008). The Rise of Computationally Enabled Economics. *Eastern Economic Journal, 34*(4), pp. 423-428.

Axtell, R. L. (2011). Zero Intelligence Traders - Non-Object-Oriented Version. Fairfax, Virginia, USA.

Axtell, R. L. (2016). 120 Million Agents Self-Organize into 6 Million Firms: A Model of the U.S. Private Sector. *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems* (pp. 806-816). Singapore: International Foundation for Autonomous Agents and Multiagent Systems.

Axtell, R. L., Axelrod, R., Epstein, J. M., & Cohen, M. D. (1996). Aligning simulation models: A case study and results. *Computational & Mathematical Organization Theory, 1*(2), 123-141.

Bankes, S. C. (2002). Agent-based modeling: A revolution? *Proceedings of the National Academy of Sciences, 99*, pp. 7199-7200.

Benthall, S. (2016). Philosophy of Computational Social Science. *The Journal of Natural and Social Philosophy, 12*(2), 13-30.

Bradhurst, R. A., Roche, S. E., East, I. J., Kwan, P., & Garner, M. G. (2016). Improving the computational efficiency of an agent-based spatiotemporal model of livestock disease spread and control. *Environmental Modelling and Software, 77*, 1-12.

Brown, N. (2015). Epiphany Basic. *GitHub*. GitHub. Retrieved from https://github.com/mesham/ebasic

Cioffi-Revilla, C. (2014). *Introduction to Computational Social Science.* New York: Springer.

Cleary, A. J., Smith, S. G., Vassilevska, T. K., & Jefferson, D. R. (2005). *Scalable Entity-Based Modeling of Population-Based Systems, Final LDRD Report.* Lawrence Livermore National Laboratory. Livermore: United States Department Of Energy. Retrieved from https://e-reports-ext.llnl.gov/pdf/315877.pdf

Cloudera. (2015a). How-to: Tune Your Apache Spark Jobs (Part 1). Retrieved from http://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-1/

Cloudera. (2015b). How-to: Tune Your Apache Spark Jobs (Part 2). Retrieved from https://blog.cloudera.com/blog/2015/03/how-to-tune-your-apache-spark-jobs-part-2/

Cordasco, G., De Chiar, R., Mancuso, A., Mazzeo, D., Scarano, V., & Spagnuolo, C. (2013). Bringing together efficiency and effectiveness in distributed simulations: The experience with D-Mason. *SIMULATION, 89*(10), 1236-1253.

Cordasco, G., De Chiara, R., Mancuso, A., Mazzeo, D., Scarano, V., & Spagnuolo, C. (2012). A Framework for Distributing Agent-Based Simulations. *European Conference on Parallel Processing* (pp. 460-470). Berlin, Heidelberg: Springer.

Crooks, A. T. (2017). Cellular automata. In D. Richardson, N. Castree, M. F. Goodchild, A. Kobayashi, W. Liu, & R. R. Marston (Eds.), *International Encyclopedia of Geography: People, the Earth, Environment and Technology.* John Wiley & Sons, Ltd.

Crooks, A. T., & Hailegiorgis, A. B. (2014). An agent-based modeling approach applied to the spread of cholera. *Environmental Modelling & Software, 62*, pp. 164-177.

Crooks, A. T., Castle, C., & Batty, M. (2008). Key Challenges in Agent-Based Modelling for Geo-Spatial Simulation. *GeoComputation: Modeling with spatial agents, 32*(6), 417-430.

Dematte, L., & Prandi, D. (2010). GPU computing for systems biology. *Briefings in Bioinformatics, 11*, 323-333.

Dongarra, J., Gannon, D., Fox, G., & Kennedy, K. (2007). The Impact of Multicore on Computational Science Software. *CTWatch Quarterly, 3*(1).

Drummond, C. D. (2009). Replicability is not Reproducibility: Nor is it Good Science. *Proceedings of the Evaluation Methods for Machine Learning Workshop at the 26th ICML*. Montreal.

Epstein, J. M. (1999). Agent-based computational models and generative social science. *Complexity, 4*(5), 41-60.

Epstein, J. M., & Axtell, R. L. (1996). *Growing Artificial Societies.* Washington: The Brookings Institution.

Epstein, J. M., & Axtell, R. L. (1997). Artificial societies and generative social science. *International Symposium on Artificial Life and Robotics. 1*, pp. 33-34. Oita: Springer-Verlag.

Fachada, N., Lopes, V. V., Martins, R. C., & Rosa, A. C. (2016). Parallelization Strategies for Spatial Agent-Based Models. *International Journal of Parallel Programming*, 1-33. Retrieved 2016

Flynn, M. J. (1966). Very high speed computers. *Proceeding of the IEEE, 54*, pp. 1901-1909.

Gardner, M. (1970). The fantastic combinations of John Conway's new solitaire game "life". *Scientific American*, pp. 120-123. Retrieved from http://www.ibiblio.org/lifepatterns/october1970.html

Geist, A., & Reed, D. A. (2015). A survey of high-performance computing scaling challenges. *International Journal of High Performance Computing Applications*, 1-10.

Gilbert, N., & Terna, P. (2000). How to build and use agent-based models in social science. *Mind & Society, 1*(1), 57-72.

Gode, D. K., & Sunder, S. (1993). Allocative Efficiency of Markets with Zero-Intelligence Traders: Market as a Partial Substitute for Individual Rationality. *The Journal of Political Economy, 101*(1), 119-137.

Gong, Z., Tang, W., Bennett, D. A., & Thill, J.-C. (2012). Parallel agent-based simulation of individual-level spatial interactions within a multicore computing environment. *International Journal of Geographical Information Science, 27*(6), 1152-1170.

Grimm, V., & Railsback, S. F. (2005). *Individual-based Modeling and Ecology.* Princeton: Princeton University Press.

Hayes, R., Todd, A., Chaidarun, N., Tepsuporn, S., Beling, P., & Scherer, W. (2014). An agent-based financial simulation for use by researchers. *Proceedings of the 2014 Winter Simulation Conference* (pp. 300-309). Savanah: IEE.

Heijnen, P., Chappin, É. J., & Nikolic, I. (2014). Infrastructure Network Design with a Multi-Model Approach: Comparing Geometric Graph Theory with an Agent-Based Implementation of an Ant Colony Optimization. *Journal of Artificial Societies and Social Simulation*. Retrieved from http://jasss.soc.surrey.ac.uk/17/4/1.html

Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming* (Revised First Edition ed.). Waltham: Morgan Kaufmann.

Hogeweg, P., & Hesper, B. (1983). The Ontogeny of the Interaction Structure in Bumble Bee Colonies: A MIRROR Model. *Behavioral Ecology and Socialbiology*(12), 271-283.

Holcombe, M., Coakley, S., & Smallwood, R. (2006). A general framework for agent-based modelling of complex systems. *Proceedings of the 2006 European conference on complex systems.*

Husselmann, A. V., & Hawick, K. (2011). Simulating species interactions and complex emergence in multiple flocks of BOIDS with GPUS. *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems* (pp. 100-107). Dallas: ResearchGate.

Kim, I.-H., Tsou, M.-H., & Feng, C.-C. (2015). Design and implementation strategy of a parallel agent-based Schelling model. *Computers, Environment and Urban Systems, 49*, 30-41.

Laville, G., Mazouzi, K., Lang, C., Philippe, L., & Marilleau, N. (2013). Using GPU for multi-agent soil simulation. *Proceedings of the 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 392-399). Belfast: IEEE.

Lazer, D., Alex, P., Adamic, L., Aral, S., Barabási, A.-L., Brewer, D., . . . Van Alstyne, M. (2009). Computational Social Science. *Science*, pp. 721-723.

Leinweber, M., Bitter, P., Bruckner, S., Mosch, H.-U., Lenz, P., & Freisleben, B. (2014). GPU-based simulation of yeast cell flocculation. *Proceedings - 2014 22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (pp. 601-608). Turin: IEEE Computer Society.

Luke, S., Cioffi-Revilla, C., Panait, L., Sullivan, K., & Balan, G. (2005). MASON: A Multi-Agent Simulation Environment. *Simulation: Transactions of the society for Modeling and Simulation International, 82*(7), 517-527.

Lysenko, M., & D'Souza, R. M. (2008). A framework for megascale agent based model simulations on graphics processing units. *Journal of Artificial Societies and Social Simulation, 11*(4), 10. Retrieved from http://jasss.soc.surrey.ac.uk/11/4/10/10.pdf

Macy, M. W., & Willer, R. (2002). From Factors to Actors: Computational Sociology and Agent-Based Modeling. *Annual Review of Sociology, 28*, 143-166.

McCabe, S., Brearcliffe, D. K., Froncek, P., Hansen, M., Kane, V., Taghawi-Nejad, D., & Axtell, R. L. (2016). A comparison of languages and frameworks for the parallelization of a simple agent model. *Proceedings of the 17th International Workshop on Multi-Agent-Based Simulation* (pp. 126-144). Singapore: Multi-Agent-Based Simulation (MABS) 2016.

McCanny, J., Ridge, J., Hu, Y., & Hunter, J. (1997). Hierarchical VHDL libraries for DSP ASIC. *Acoustics, Speech, and Signal Processing (ICASSP-97)* (pp. 675-678). IEEE.

Meeth, L. R. (1978). Interdisciplinary Studies: A Matter of Definition. *Change, 10*(7), 10.

Moore, G. E. (1965). Cramming more components onto integrated circuits. *Electronics, 38*(8).

North, M. J., Collier, N. T., Ozik, J., Tatara, E. R., Macal, C. M., Bragen, M., & Sydelko, P. (2013). Complex adaptive systems modeling with Repast Simphony. *Complex Adaptive Systems Modeling, 1*(3).

Nurvitadhi, E., Sim, J., Sheffield, D., Mishra, A., Krishnan, S., & Marr, D. (2016). Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. *26th International Conference on Field Programmable Logic and Applications (FPL)* (pp. 1-4). Lausanne: IEEE.

NVIDIA Corporation. (2012). *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110.* Santa Clara: NVIDIA Corporation. Retrieved from https://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

NVIDIA Corporation. (2013). Unified Memory in CUDA 6. Santa Clara. Retrieved from https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/

NVIDIA Corporation. (2017a). CUDA Parallel Computing Platform. Santa Clara. Retrieved from http://www.nvidia.com/object/cuda_home_new.html

NVIDIA Corporation. (2017b). Unleash Your Potential with the Jetson TX1 Developer Kit. Santa Clara. Retrieved from https://developer.nvidia.com/embedded/buy/jetson-tx1-devkit

NVIDIA Corporation. (2017c). Accelerated Computing - Training. Santa Clara. Retrieved from https://developer.nvidia.com/accelerated-computing-training

NVIDIA Corporation. (2017d). What is GPU-Accelerated Computing? Santa Clara. Retrieved from http://www.nvidia.com/object/what-is-gpu-computing.html

NVIDIA Corporation. (2017e). Parallel Thread Execution. Santa Clara.

NVIDIA Corporation. (2017f). An Even Easier Introduction to CUDA. Santa Clara. Retrieved from https://devblogs.nvidia.com/parallelforall/even-easier-introduction-cuda/

Olofsson, A. (2016). *Epiphany-V: A 1024 processor 64-bit RISC System-On-Chip.* Lexington: Adapteva Inc. Retrieved from https://www.parallella.org/wp-content/uploads/2016/10/e5_1024core_soc.pdf

Ostrom, T. M. (1988). Computer simulation: The third symbol system. *Journal of Experimental Social Psychology*, 381-392.

Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., & Phillips, J. C. (2008). GPU Computing. *Proceedings of the IEEE. 96*, pp. 879-899. IEEE.

Papadimitriou, C. H., & Yannakakis, M. (1994). On complexity as bounded rationality. *STOC '94 Proceedings of the twenty-sixth annual ACM symposium on Theory of computing* (pp. 726-733). New York: ACM.

Parunak, H. V., Savit, R., & Riolo, R. L. (1998). Agent-Based Modeling vs. Equation-Based Modeling: A Case Study and Users' Guide. *Proceedings of Multi-agent systems and Agent-based Simulation (MABS'98)* (pp. 10-25). Paris: Springer.

Railsback, S. F., & Grimm, V. (2012). *Agent-Based and Individual-Based Modeling: A Practical Introduction.* Princeton: Princeton University Press.

Ralston, A., & Meek, C. (Eds.). (1976). *Encyclopedia of Computer Science.* New York: Litton Educational Publishing, Inc.

Ren, C., Yang, C., & Jin, S. (2009). Agent-Based Modeling and Simulation on Emergency Evacuation. *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, 5*, pp. 1451-1461.

Reynolds, C. W. (1987). Flocks, herds and schools: A distributed behavioral model. *SIGGRAPH '87 Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (pp. 25-34). New York: ACM SIGGRAPH Computer Graphics.

Richie, D., Ross, J., Park, S., & Shires, D. (2015). Threaded MPI programming model for the Epiphany RISC array processor. *Journal of Computational Science, 9*, 94-100.

Šalamon, T. (2011). *Design of Agent-Based Models.* Živonín: Tomáš Bruckner.

Schelling, T. C. (1969). Models of Segregation. *The American Economic Review, 59*(2), 488-493.

Schelling, T. C. (1971). Dynamic Models of Segregation. *Journal of Mathematical Society, 1*, 143-186.

Shook, E., Wang, S., & Tang, W. (2013). A communication-aware framework for parallel spatially explicit agent-based models. *International Journal of Geographical Information Science, 27*(11), 2160-2181.

Simon, H. A. (1955). A Behavioral Model of Rational Choice. *The Quarterly Journal of Economics, 69*(1), 99-118.

Simon, H. A. (1996). *The Sciences of the Artificial* (Third ed.). Cambridge: The MIT Press.

Swarm Development Group. (1999). Swarm. Santa Fe, New Mexico: Swarm Development Group. Retrieved November 11, 2016, from http://www.swarm.org

Tsang, E. P., & Martinez-Jaramillo, S. (2004). Computational Finance. *IEEE computational intelligence society newsletter, 3(8)*. IEEE.

Uchmański, J., & Grimm, V. (1996). Individual-based modelling in ecology: what makes the difference? *Trends in Ecology and Evolution, 11*(10), 437-441.

Walker, D. C., Hill, G., Wood, S. M., Smallwod, R. H., & Southgate, J. (2004). Agent-based computational modeling of wounded epithelial cell monolayers. *NanoBioscience, IEEE Transactions on, 3*(3), pp. 153-163.

Wang, D., & Huang, J. (2015). databricks. *Tuning Java Garbage Collection for Apache Spark Applications*. Retrieved from https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html

Wang, J., Rubin, N., Wu, H., & Yalamanchili, S. (2013). Accelerating simulation of agent-based models on heterogeneous architectures. *ACM International Conference Proceeding Series* (pp. 108-119). Houston: ACM.

Watts, D. J. (2013). Computational social science: Exciting progress and future directions. *The Bridge on Frontiers of Engineering, 43*(4), 5-10.

Wendel, S., & Dibble, C. (2007). Dynamic Agent Compression. *Journal of Artificial Societies and Social Simulation, 10*(2), 9. Retrieved from http://jasss.soc.surrey.ac.uk/10/2/9.html

Wilensky, U. (1999). NetLogo. Evanston: Center for Connected Learning and Computer-Based Modeling, Northwestern University. Retrieved from http://ccl.northwestern.edu/netlogo/

Wilensky, U., & Rand, W. (2015). *An Introduction to Agent-Based Modeling.* Cambridge: The MIT Press.

Xiong, C. (2015). *On Agent-Based Modeling: Multidimensional Travel Behavioral Theory, Procedural Models and Simulation-Based Applications.* College Park: University of Maryland. Retrieved from http://drum.lib.umd.edu/handle/1903/17362

Yang, C., Ono, I., Kurahashi, S., Jiang, B., & Terano, T. (2015). A grid based simulation environment for parallel exploring agent-based models with vast parameter space. *Lecture Notes in Computer Science* (pp. 534-548). Springer Verlag.

Zaho, T., Li, T., Han, B., Sun, Z., & Huang, J. (2014). Design of Software Defined hardware counters for SDN. *2014 IEEE 20th International Workshop on Local & Metropolitan Area Networks (LANMAN)* (pp. 1-6). Reno: IEEE.

BIOGRAPHY

Dale K. Brearcliffe is a Masters student in Computational Social Science at George Mason University. He attended California State University Hayward (now known as California State University East Bay), where he received his Bachelor of Science in Computer Science in 1983. He is currently a senior consultant and data scientist.