EXPLOITING IMMUNOLOGICAL METAPHORS IN THE DEVELOPMENT OF SERIAL, PARALLEL, AND DISTRIBUTED LEARNING ALGORITHMS

a thesis submitted to

The University of Kent

in the subject of computer science

for the degree

of doctor of philosophy.

By

Andrew B. Watkins

March 2005

© Copyright 2005

by

Andrew B. Watkins

Abstract

This thesis examines the use of immunological metaphors in building serial, parallel, and distributed learning algorithms. It offers a basic study in the development of biologically-inspired algorithms which merge inspiration from biology with known, standard computing technology to examine robust methods of computing. This thesis begins by detailing key interactions found within the immune system that provide inspiration for the development of a learning system. It then exploits the use of more processing power for the development of faster algorithms. This leads to the exploration of distributed computing resources for the examination of more biologically plausible systems.

This thesis offers the following main contributions. The components of the immune system that exhibit the capacity for learning are detailed. A framework for discussing learning algorithms is proposed. Three properties of every learning algorithm—memory, adaptation, and decision-making—are identified for this framework, and traditional learning algorithms are placed in the context of this framework. An investigation into the use of immunological components for learning is provided. This leads to an understanding of these components in terms of the learning framework. A simplification of the Artificial Immune Recognition System (AIRS) immune-inspired learning algorithm is provided by employing affinity-dependent somatic hypermutation. A parallel version of the Clonal Selection Algorithm (CLONALG) immune learning algorithm is developed. It is shown that basic parallel computing techniques can provide computational benefits for this algorithm. Exploring this technology further, a parallel version of AIRS is offered. It is shown that applying these same parallel computing techniques to AIRS, while less scalable than when applied to CLONALG, still provides computational gains. A distributed approach to AIRS is offered, and it is argued that this approach provides a more biologically appealing model.

Biological immune systems exhibit complex cellular interactions. The mechanisms of these interactions, while often poorly understood, hint at an extremely powerful information processing/problem solving system. This thesis demonstrates how the use of immunological principles coupled with standard computing technology can lead to the development of robust, biologically-inspired learning algorithms.

Dedication

For Molly.

Acknowledgments

In producing this work, I have incurred a great debt of gratitude to the following:

Jon Timmis has been much more than a great supervisor: he's been a great friend. From shuttles to and from the airport to week- and month-long stays at his house, from transatlantic phone calls to deliciously cooked meals, Jon has provided so much more than timely academic guidance. And while I blame him entirely for me ever having attempted this work, I know that I am glad for the persuasion and happy to have been through it with him.

The Computing Laboratory at the University of Kent agreed to fund this strange PhD arrangement. Without their willingness to try out this experiment, I would never have completed this degree.

The Department of Computer Science and Engineering at Mississippi State University has been invaluable in the development and fruition of this work. Not only did they provide a full-time teaching job for me so that I could eat and yet have just enough time to work on this research, they also provided computing platforms and system support for many of my experiments. In particular, I would like to thank Drs. Lois Boggess, Susan Bridges, Yogi Dandass, Julia Hodges, and Donna Reese for all of their support.

Finally, I would like to thank all of those I have interacted with in the AIS community whose ideas and discussions have made this a better work. In particular, I'd like to thank the participants at WCCI '02 in Hawaii, at the ICARIS conferences, and of the Kent AIS reading group. I would also like to thank the anonymous reviewers of the various papers that have been produced based on this work. Their comments have been extremely useful in shaping and directing this research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation and Background

1.1.1 Intelligent Machines

Let us start with the grandiose: Since the conceptualization of a computing machine, we have been asking if we can get these machines to think—if they are capable of displaying intelligence. Indeed, even, arguably, the founder of computer science, Alan Turing, was intrigued by this concept—so much so, that we have, as one of his many legacies, the Turing test which provides us a litmus test to determine if our efforts to impersonate or duplicate human intelligence have been successful through deceiving someone into thinking the machine is human. While many of today's researchers view the Turing test as nothing but a useless distraction, it does indicate how our imaginations have been captured by this dream of thinking machines.

Wherein lies the impetus for this quest? One could argue that there are really two thrusts to this idea. One, the more traditional and, perhaps more mundane, is that computers are seen as problem solving tools—computational wizards, so to speak. And, if we could harness the sheer computational power of the computer and couple with it the ability to think—to conceptualize, reason, and generalize—found in human intelligence then the realm of problems to be conquered by these thinking machines, the aid that could be given to humans by this ability, is close to limitless. The more philosophical reason is that, as in many human endeavors, we are trying to more fully understand ourselves. This exploration of what is our own intelligence has simply taken on as a new experimental test bed the digital computer. That is, if we can succeed in embedding something close to human intelligence into these bits of silicon and electricity, then we can actually discover more about ourselves and about the ways our cognitive abilities work.[1] For now we will focus on the more mundane of these two answers—that is, we will focus on the idea of building problem solving tools and aids to our own intelligence. Yet, the larger answer should, perhaps, always be in the back of our minds as we explore this tool building exercise.

[1]We do not mean by this only simulations which are attempts to be faithful replicas of the biological processes involved in intelligence. While this computer modeling is definitely useful, what we have in mind here as the more grandiose side of the question is that even through our building intelligent machines for problem solving we have to contemplate and discover how it is that our cognitive systems are able to tackle such problems, and through this exploration—even when building problem solvers—we gain a deeper insight into our own intelligence.

1.1.2 Machine Learning

The ability to improve with experience, to get better over time, to remember past decisions and outcomes in order to make better choices in future, similar situations, that is, the ability to learn can be seen as a fundamental characteristic of human intelligence. This being said, it is little surprise that so much research in the field of artificial intelligence (AI) could be labeled, in one way or another, as research into machine learning. What, then, is machine learning? To quote Tom Mitchell, "The field of machine learning is concerned with the question of how to construct computer programs that automatically improve with experience" [86]. We will accept this definition as good as any other for now concerning this field. However, while we may have a working definition of what machine learning is, we do not, necessarily, have a reason as to why we would want such a thing. Why do we care to develop programs that can "automatically improve with experience?" The answer to this can be seen as the same as the answer to why we would want to develop intelligent machines at all: aid in problem solving and exploration of human intelligence.

Continuing with our more pragmatic slant, then, the development of computer programs that improve with experience should be triggered by the ability this gives us to solve problems of interest. So how do we characterize these problems? The answer to this question is large and will be, by no means, treated comprehensively here. However, to point towards an answer we can look at just a few anecdotes:

- We have a large amount of data collected from sonar scans of the ocean floor. Geological and oceanographic experts have classified some of this data into categories of interest. However, given the volume of data, there is no way possible for these experts to examine and classify all of the data. We would like to develop a computer program that can automatically learn from the classified data set the characteristics of the data that are of interest and then classify the remaining set of acoustic images.

- We have a database of consumer transactions. We would like to discover if there are any patterns of buying certain products together so that we can better market products of interest to consumers. Again, given the sheer volume of data, there is no practical way for a human (or group of humans) to examine all of this data. Also, there potentially are buying habits that humans would never have considered and thus never looked for. We want to develop a computer program that can examine this data and automatically learn/discover the patterns of purchasing habits that the database contains. We can then use this information for more informed product marketing/recommendations.

- We want to develop a robot that can empty the trash for us. It should be able to wander around our house, empty full trash bins, and remain unobtrusive. In addition, it should know when it is about to run out of battery power and recharge itself. This would require a system that can learn how to navigate, learn time management (so that it does not run out

of power or allow a trash bin to overflow), and learn to remain unobtrusive. We would expect it to get better at its job the longer it was in service.

These are just three simple examples of the types of problems that machine learning techniques are devised to address. In the parlance of machine learning we could characterize these three problems as supervised learning, unsupervised learning, and reinforcement learning, respectively, although these labels have more to do with the solution rather than a characteristic of the problem itself.

Having, however briefly, looked at the what (using Mitchell's definition) and the why (the solving of certain types of problems) of machine learning, we will now turn to the how of machine learning. How do we actually develop these programs that learn to "automatically improve with experience?" This will be the primary focus of the rest of this thesis. However, the what and why will always be in mind as we go through the construction of such systems. The means of constructing and developing machine learning systems has been extremely diverse. These have ranged from the manipulation of symbolic data in order to develop a concept learning system to the heavy use of sub-symbolic representations of data in the development of function approximation algorithms (e.g., back-propagation neural networks). There are several good general surveys of the field of machine learning and its techniques (foremost being [86]), and we will not attempt to duplicate that effort here. Instead, we wish to focus on one particular emerging field of computing science that can, perhaps, lend insight into the development of machine learning systems.

1.1.3 Biologically-Inspired Computing

When attempting to build more complex systems, systems that resemble the intelligence or efficiency found in natural systems, it is not surprising that computer scientists have often turned to biological systems for inspiration in solving complex computational problems. The landscape of computation is littered with biologically-inspired models. These range from simple neural network models derived from observations of the brain[12, 58] to computational explorations based on neo-Darwinian evolutionary theory[85]; from capturing characteristics of swarming insects for computation[70] to extracting inspiration from the behavior of ants[14], to name just a few. While the problems these biologically-inspired systems tackle are as diverse as the natural systems that inspired them, they all share common links. Namely, they all attempt to extract mechanisms from nature that can lend themselves to the tackling of computational problems or to the further understanding of the nature of computation itself. While this often leads to a fairly naïve examination of the biology (as pointed out in [108]), the success of such systems is no doubt in large part attributable to the biological inspiration at the heart of their development.

In [108] the authors argue for the need for a well-formed conceptual framework for the development of biologically-inspired computing systems. This thesis provides an in-depth examination of the development of a bio-inspired system. While this process may often fall into the trap of "reasoning by metaphors" that the authors of [108] warn against, it does provide a solid illustration of the give

and take seen in the development of such systems. Throughout this development process we will see an almost pendulum-like swinging back and forth as we first look to the biology for inspiration and then incorporate established standard computational technologies and then return to the biology only to incorporate other standard computing techniques. This is one of the more intriguing aspects of the empirical development of a biologically-inspired computing system: the ways in which biology and standard computing can inform and transform the developing computational system.

1.1.4 Immune-Inspired Computing

The immune system is a robust, complex collection of diverse, interacting components that serve to protect the body. It consists of multiple-layers of response and numerous reactions distributed throughout the body. Simplistically, in vertebrates, the immune system is composed of two layers: the innate immune system and the adaptive immune system. The innate system is a fairly static first line of defense. It consists of cells that recognize incoming antigens and presents pieces of these invaders to the adaptive system. The adaptive immune system is an ever-evolving dynamic system that reacts and interacts with the environment in a variety of ways. Much of the way the immune-system reacts and behaves is poorly understood. However, what we do understand offers tantalizing possibilities from a computational point of view.

The biological immune system is diverse, inherently distributed, adaptive, complex, capable of maintaining memory of previous encounters, robust, and interactive, to name just a few of its more attractive computational properties. In the last fifteen years there has been a great deal of interest in exploiting the known properties of the immune system as metaphorical inspiration for computational problem solving. Applications of these techniques have ranged from computer security [62, 72] to robotics [78, 87] to data analysis and machine learning [113, 32, 130]. It is this last area of immune-inspired computing—machine learning—that is the focus of our work here.

1.1.5 Learning in Parallel and Distributed Environments

Beyond just the development of learning algorithms inspired from observations of nature, a second major theme of this work is the ability to harness greater computational power for these tasks. With the recent proliferation of clusters of computers due to the decreasing costs of commodity computing components, it has now become extremely feasible to dedicate multiple computers to a given problem solving task. This leads to several possible consequences. The most obvious one is the ability to speed up the overall processing time in order to arrive at more timely solutions. This is extremely appealing from a machine learning/data mining perspective. As we previously mentioned, one of the motivations for developing machine learning algorithms is in their abilities to save time for humans in complex tasks. By utilizing multiple computers or large parallel processors,

machine learning algorithms, which can take advantage of this increased power, will also increase their benefits to the user.

This use of multiple processes in our learning algorithm also provides the ability for the development of more robust, or in depth, solutions. In other words, we can take the known computational technology of a parallel system and look for ways to incorporate this power in our developing biologically-inspired system. With many bio-inspired algorithms taking on an evolutionary component, the ability to evolve and explore separate niches and species of solutions on individual processors and then bring these individual populations to bear on the problem as a whole is potentially invaluable. The use of these distributed processing techniques allows us to further enhance our immune model. The immune system is inherently distributed and decentralized; therefore, it is important to the evolution of the field of artificial immune systems that we explore the use of distributed and parallel computing in order to more fully explore this biological potentials. While there has been some work on machine learning and evolutionary algorithms using large multi-computers or clusters of computers (see [17, 22] for some basic examples), very little in the application of parallel and distributed computing techniques for immune-inspired learning has been explored.

1.2 Goals and Contributions

The overarching hypothesis of this thesis is that the identification and exploitation of immunological components coupled with standard computing techniques can

lead to the development of serial, parallel, and distributed learning algorithms. In order to investigate this hypothesis we begin by examining the fields of immunology and machine learning. Through this we identify those characteristics of the immune system that are most pertinent to improving with experience. We then survey previous work in the field of immune-inspired learning algorithms. This leads to the presentation of one specific immune learning algorithm, which we examine in depth. This examination provides an illustrative example of the development of a biologically-inspired system. We then look to incorporate more standard computing technology by examining how parallel computing techniques can be used in this area of immune-inspired learning. This leads to the development of parallel versions of two artificial immune systems: one as a proof-of-concept of the use of the technology and the second as a return to our initial exemplar system. We end our examination with a look at a more distributed approach to the problem and argue that this is the path forward toward a more biologically plausible and interesting approach to immune-inspired computing.

Through these investigations, this thesis makes the following contributions.

1. The components of the immune system that exhibit the capacity for learning are detailed (chapter 2).

2. A framework for discussing learning algorithms is proposed. Three properties of every learning algorithm—memory, adaptation, and decision-making—are identified for this framework, and traditional learning algorithms are placed in the context of this framework (chapter 2).

3. An investigation into the use of immunological components for learning is provided. This leads to an understanding of these components in terms of the learning framework (chapter 2).

4. A simplification of the AIRS immune-inspired learning algorithm is provided by employing affinity-dependent somatic hypermutation (chapter 3).

5. A parallel version of the CLONALG immune learning algorithm is developed. It is shown that basic parallel computing techniques can provide computational benefits for this algorithm (chapter 5)

6. Exploring this technology further, a parallel version of AIRS is offered. It is shown that applying these same parallel computing techniques to AIRS, while less scalable than when applied to CLONALG, still provides computational gains (chapter 6).

7. A distributed approach to AIRS is offered, and it is argued that this approach provides a more biologically appealing model. The simple distributed approach is proposed in terms of an initial step toward a more complex, distributed system (chapter 7).

1.3 Thesis Structure

The remainder of this thesis is organized into the following seven chapters.

Chapter 2 surveys the three main fields pertinent to the first part of this study: immunology, machine learning, and artificial immune systems. All three of these

are too vast to be treated comprehensively. This survey chapter identifies those components of the immune system that are most relevant for developing learning algorithms. We then propose a framework for discussing learning algorithms and place several well-known algorithms into this framework. We conclude the chapter by surveying previous immune-inspired learning algorithms and discussing them in terms of our framework.

Chapter 3 introduces the immune-inspired learning algorithm AIRS as our exemplar system to be studied throughout the thesis. The chapter begins with a brief overview of the algorithm and previous results obtained by using this algorithm. We then offer a series of simplifications to the algorithm designed to incorporate the more biologically sound mutation scheme of affinity-dependent somatic hypermutation. We find that this change decreases the overall memory model of the algorithm while maintaining classification accuracy. We conclude the chapter by comparing the original and newly formulated AIRS algorithm.

Chapter 4 provides a brief overview of parallel processing techniques and performance metrics. It briefly surveys parallel genetic algorithms and discusses how these parallel processing ideas can be applied to immune learning algorithms.

Chapter 5 examines the application of parallel computing techniques to a basic immune-inspired learning algorithm: CLONALG. We examine the behavior of this new parallel version of the algorithm. We find that there are definite computational gains to be made by parallelizing CLONALG.

Chapter 6 presents a parallel version of AIRS. We begin by following the same strategies employed with parallelizing CLONALG. However, unlike CLONALG

which has no real global interaction, AIRS does require global communication in order to build its memory model. We investigate several ways to handle the need for developing a single memory cell pool. This leads to the identification of the memory cell pool as the primary bottleneck in the parallel process. Despite unstable parallel performance, we do find benefits from our basic parallelization of AIRS.

Chapter 7 returns to the biology of the immune system and offers an alternative approach to using multiple processors for AIRS. Since the immune system is not centralized but distributed, we would like to develop algorithms that also exhibit this characteristic. This chapter provides a very simple distributed model for AIRS. While the results using this approach are somewhat inconclusive, this study raises interesting possibilities for the development of distributed immune learning algorithms.

Chapter 8 provides concluding remarks for this study. It also points the way forward to future extensions of this work.

1.4 Publications

The following four papers were written as part of the research for this thesis:

1. A. Watkins and J. Timmis, "Artificial immune recognition system (AIRS): Revisions and refinements," in *Proceedings of the 1st International Conference on Artificial Immune Systems (ICARIS2002)*, J. Timmis and

P. J. Bentley, Eds. University of Kent at Canterbury: University of Kent at Canterbury Printing Unit, September 2002, pp. 173-181.

2. A. Watkins, X. Bi, and A. Phadke, "Parallelizing an immune-inspired algorithm for efficient pattern recognition," in *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems and Artificial Life*, C. Dagli, A. Buczak, J. Ghosh, M. Embrechts, and O. Ersoy, Eds. New York: ASME Press, November 2003, vol. 13, pp. 225-230.

3. A. Watkins and J. Timmis, "Exploiting parallelism inherent in AIRS, an artificial immune classifier," in *Proceedings of the 3rd International Conference on Artificial Immune System (ICARIS2004)*, ser. Lecture Notes in Computer Science, G. Nicosia, V. Cutello, P. Bentley, and J. Timmis, Eds., no. 3239. Springer-Verlag, September 2004, pp. 427-438.

4. A. Watkins, J. Timmis, and L. Boggess, "Artificial immune recognition system (AIRS): An immune-inspired supervised machine learning algorithm," *Genetic Programming and Evolvable Machines*, vol. 5, no. 3, pp. 291-317, September 2004.

Chapter 2

Biological Immune Systems and Learning

This chapter examines the background for building immune inspired learning systems. We begin by examining key components in biological immune systems. This survey is, naturally, biased towards those components that indicate learning; however, it should give a basic comprehension of the biology necessary for understanding the rest of this thesis. This is followed by the introduction of a generic framework for learning systems as well as a discussion of some well-known learning algorithms within this framework. This framework is introduced as a means of quickly comparing or understanding the fundamental workings of the learning algorithms we will examine. We conclude this chapter by surveying immune-inspired learning algorithms. This section provides a context for the work to come.

2.1 Biological Immune Systems

Intuitively when we think of the immune system the idea of learning or any type of cognition does not readily spring to mind. The immune system is seen as a great protector—warding off those unseen microscopic organisms that can cause death and destruction to the body, mechanically going about the business of defense statically as it was designed to do. Where would be the learning in this? Yet, as we consider it more fully, we realize that we have all had experiences with this idea of the immune system learning and improving over time. The concepts of inoculation or vaccination are immediate examples. How is it that by exposing the immune system to a small dose of a disease it is later able to protect us completely from exposure to that disease? How is it that the immune system is capable of rapidly defeating previously seen and even not seen, but similar, pathogens? When seen in this way we come to realize that at the very least there must be some type of memory mechanism involved in the immune system, and there are hints just from these practical every-day experiences that the immune system must indeed be learning its primary job of protecting the body (if indeed this truly is the primary job of the immune system). So what is it in the immune system that enables it to learn from interactions with the environment and provide appropriate responses when faced with the same or similar situations (pathogens)? Theoretical and experimental immunology points to a wealth of possible answers.

2.1.1 Immunological Components

The potential complexity of the immune system is staggering at times, and its workings are still poorly understood to a large degree. Some things that do seem to be understood is that the immune system can be viewed as a layered system. The first line of defense, described as innate immunity, begins at the skin-level and delves further to include several pattern recognition receptors that process incoming pathogenic patterns for presentation to other layers of the immune system. The pattern recognition capabilities of these innate cells are fairly fixed throughout the lifetime of an organism [84]. So, while they play a key role in any immune response and (perhaps) could be a useful source of inspiration for one stage in a machine learning system, since there is really no adaptation nor change going on at this stage with regards to the interaction with the environment—no real learning that is—we will not spend any more time on this level.

Lymphocytes

Typically, where we think of most of the learning and adapting of the immune system taking place is in the, aptly named, adaptive immune response. Again, there is a wealth of complexities here, but what we will primarily focus on are the lymphocyte cells–T-cells and B-cells. The two cells get their names from where they mature (thymus and bone-marrow, respectively). What is important in the context of our work about these cells is the ability of these two types of cells to interact and change in response to incoming pathogens. (At this point we will

just switch, out of simplicity, to calling them antigens, since it is the antigenic patterns of a given pathogen that are really of interest to us at the T- and B-cell level).

Since the immune cells must be able to know not to attack the body but to attack antigens, there must be some mechanism for imparting this knowledge at the cellular level. For the T-cells this is done through a maturation and negative-selection process. Naïve T-cells in the thymus are repeatedly exposed to self-cells. Those T-cells which react to the self-cells are killed off. However, after a certain amount of this exposure, if the T-cell does not react to the self-cells, then it is allowed into the body as a mature T-cell. One theory is that through this process, termed negative-selection, we have an immune system with T-cells that, when they do react, should only be reacting to harmful (or at least non-self) antigens [97, 68, 119]. Similar to this idea, yet fundamentally different, is the so-called Danger Theory idea of T-cell response [83]. This theory states that the idea of "self" and "non-self" causing reactions in T-cells is ludicrous since there are numerous "non-self" agents that are tolerated by the body (e.g., food in the gut, a fetus in the womb, etc.). Rather, T-cells learn to respond to that which is dangerous and not dangerous. This is partially accomplished through the use of "danger signals" which trigger responses in the T-cells. Regardless of which school of thought one subscribes to, however, T-cells are still involved with pattern recognition.This is one line of pattern recognition and learning that takes place in the immune system. Through this recognition process, T- Cells (or more specifically, helper T-cells at this point), in turn, will present antigenic patterns

to B-cells, which will also react in specific ways to the antigen and proliferate in order to defend the body against the incoming invader [103, chap2, pp21-36].

We now examine the other major lymphocyte cells, B-cells. As previously mentioned, B-cells develop in the bone-marrow and interact with T-cells in order to produce immune responses. On the surface of each B-cell is a collection of structurally/chemically identical antibodies. It is the antibodies that bind to antigenic patterns and produce a reaction based on the degree of recognition in response to a given antigen [96, 103, chap5, pp80-107]. The manner in which this recognition is performed and the reaction of B-cells in response to this recognition will form the nucleus for much of our discussion involving learning and the immune system.

In the theories of the workings of the immune system, several different proposals have been made as to how a given B-cell interacts with an antigen and how this interaction affects the rest of the system. One such idea, named the Clonal-Selection theory [16], proposes that as an antigen enters the immune system certain B-cells are selected based on their reaction to this antigen to undergo rapid cloning and expansion. As an antigen is presented to a B-cell, the B-cell's antibodies react in some degree to the antigen.[1] This reaction is often termed the affinity of that B-cell (or antibody) for the given antigen. Those B-cells with a sufficient affinity (based on some internal reaction threshold) are allowed to

[1]We will see that we often talk about antibodies/B-cells in terms of straight pattern recognition with concepts of distance or complementariness; however, in reality, B-cells recognize structure as well. There has been little done to capture this idea of structural, 3-dimensional recognition rather than using simple feature distance metrics.

produce offspring in relation to their degree of affinity. This allows for the rapid expansion of cells that can successfully attack the incoming antigen.

More specifically, when a B-cell recognizes an antigen it requires stimulation from a T-helper cell to react. This co-stimulation then transforms the B-cell into a plasma cell which produces a concentration of free-antibodies which work with T-killer cells to neutralize the antigenic threat. Once the threat subsides, certain B-cells which were highly stimulated by the antigen are selected to become long-lived memory cells which allow for a more rapid secondary response if a similar antigen is encountered in the future [68, 96, 119, 103, chap10, pp177-199]. This idea of immunological memory will be discussed in section 2.1.2.

Shape-Spaces

At this point, before we continue with the dynamics of B-cell/antibody expansion and maturation, we should mention the concept of an immune system repertoire and coverage by the immune system [39, 100]. In simplistic fashion, the realm of antigen and antibody reaction can be thought of as an abstract space or volume. The attributes of an antigen or antibody, such as the presence or absence of certain chemical chains, its DNA sequence, etc., can be said to define coordinates within this space. So we can conceptualize antigens and antibodies/B-cells as occupying a given point in this space. This space is often referred to as the shape-space of the immune system. When discussing this shape-space and the realm of antibody recognition of an antigen, we claim that there is a sphere of influence or recognition

that emanates from each antibody/B-cell.[2] An antigen that is located, based on its shape-space coordinates, within the recognition sphere of a given antibody/B-cell is said to be recognized, to some degree, by this antibody/B-cell. Conceptually, then, there is a certain threshold for each antibody/B-cell that defines what its sphere of recognition will be. Figure 2.1.1 gives a graphical representation of this with three cells and their area of recognition depicted along with antigens that fall within and without these cells' areas of influence. This allows for an antibody/B-

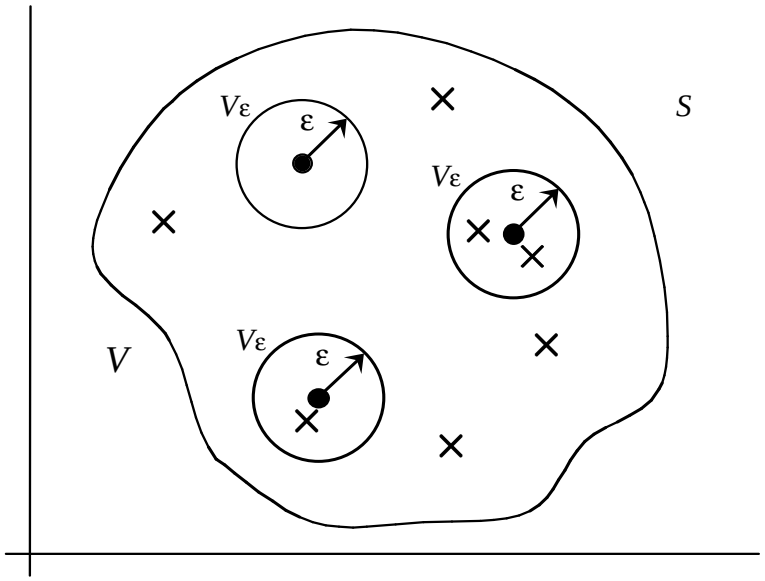

Figure 2.1.1: The shape-space concept. This image has been taken from [28].

cell to partially match an antigen and still effectively react to this threat. Also, it allows us to discuss the idea of an immune repertoire covering this shape space. That is, there is no need for a direct one-to-one antibody-to-antigen match in

[2]We are using the term "sphere" here rather loosely. As the authors of [57] point out, the actual shape of this region of influence can have an impact on the behavior of the system.

order for the immune system to appropriately respond to an antigen. In fact, this idea is quite absurd. The number of possible antigens is outrageously large whereas the human immune system has a finite number of immune cells at any given time. Yet, this finite number of cells is still capable of effectively recognizing and destroying dangerous antigens. We argue that the immune system must be able to generalize to similar patterns and to embody this idea of shape-space coverage and spheres (or volumes) of recognition in order for it to perform its duties.

Antibody-Antigen Reactions

With this discussion of shape-space and coverage in mind, we still must point out that just because a given antigen falls within the sphere of recognition of a given B-cell does not mean that it will have the same reaction as a second antigen that also falls within this sphere. That is to say, there must also be the concept of a degree of affinity to the antigen that, even within the sphere of recognition, produces different strengths of reactions, such that an antigen that falls closer to an antibody within the shape-space will elicit a stronger reaction than one which, while still within this recognition ball, is further away from the actual antibody center of this sphere. This leads naturally to the concepts of affinity maturation and somatic hypermutation [71]. As previously mentioned, B-cells are selected, based on their affinity to an antigen, to produce a number of clones. The purpose of this process is two-fold. First these clones are produced in order to attack or neutralize the invading antigen. Those cells that have a higher

degree of affinity are allowed to produce more clones since these cells will be able to more appropriately respond to the invading antigen. The second purpose of clonal production is to develop immune cells that are more adept at recognizing and reacting to the antigen. This is done through a process of affinity maturation through mutation. Each offspring of a B-cell can potentially be a mutated variety of its progenitor. Mutation rates within the immune system have been found to be quite high for some cells and are definitely variable [118]. B-cell offspring undergo mutation based in inverse proportion to their affinity values. That is, those cells which have a higher affinity value (and thus are better at recognizing the given antigen) mutate less than those cells with a lower affinity value. Through this process, the affinity of subsequent generations of B-cells will be greater (will have matured) in reaction to the antigen, and more diversity will also have been added to the system through the wider exploration afforded by the high mutation rates of the cells with lower affinity measures.

Up to this point, by the way we have described B-cell clonal expansion and reaction to an antigen, it would appear that the immune system would grow wildly out of control with large numbers of mutated clones. However, just as there are high mutation rates and cloning rates present within B-cells, there is also a high degree of cell death [68]. Given the nature of mutations, it is to be expected that a large number of these cells would not be very useful in binding to the antigen in question [4]. In fact, many of these mutations might be harmful to the self as well. The immune system, however, is self-regulating. The number of immune cells in the system at any given time, while always fluctuating, remains roughly

constant. Those B-cells which have low affinities are frequently culled from the system through the process, more than likely, of apoptosis (programmed cell-death). The exact mechanisms for this cell death are not completely understood, but it is known that there is a high turnover rate among the cells in the immune system, and it is theorized that those cells that are chosen to suffer apoptosis are the low-affinity cells [103, chap7, pp129-146].

2.1.2 Immune Memory and Learning

While this description explains at a high level how B-cells react at a single point in time to an incoming antigen, none of this really explains how, when presented with the same (or similar) antigen, the immune system responds much more efficiently and rapidly. This was one of our intuitive arguments that the immune system must possess some type of learning mechanism: through vaccination with a small dose of a disease (pathogen), the immune system is capable of protecting us from that disease. How, then, is this memory maintained? Why is the ability to efficiently react to a pathogen not lost through cell death after the initial exposure to a given antigen? The answers lie in (possibly) two directions. First, the more accepted notion is that, after the initial exposure of the immune system to a given antigen, a long-lived memory cell is maintained in the system. Upon subsequent exposure, this memory cell is allowed to produce clones much more rapidly than typical B-cells in reaction to the antigen or structurally similar antigens. In some ways, then, one of the purposes (apart from actually neutralizing the threat presented by an

antigen) of clonal selection, affinity maturation, and somatic hypermutation is the development of a memory cell that will allow for rapid responses in the future. So, immune memory is maintained through a collection of memory cells [107]. This pool of memory cells also evolves over time in reaction to the environment, but this evolution, in many ways, is on a slower scale than for typical immune cells. There seems to be less rapid cell death within the collection of memory cells. Indeed, as most of us have experienced, for many diseases one simple inoculation lasts our entire lifetimes [107].

The second possible location of immune memory is more controversial in theoretical immunology circles. Without discounting this concept of memory cells in the immune system, the second location of immune memory is within the interactions of the immune cells themselves. Jerne has proposed that B-cells are connected to each other through an idotypic network [69]. Each B-cell is connected to a neighborhood of other B-cells. These connections provide both stimulation for the cells within the neighborhood (thus keeping them alive) and suppression. There is a fine degree of balancing among the cells, and those cells which do not receive enough stimulation eventually die. When an antigen is presented to the immune network, an increase of the stimulation/suppression signals among the network of cells occurs with the neighborhood of cells that have a higher affinity to the antigen receiving more stimulation and those with less affinity receiving more suppression. This constant balancing of network connections through constant stimulation and suppression is possibly one place that immune memory also resides.

2.1.3 Diversity and Distributedness

The interactions between T-cells and B-cells and between T-cells and other immune cells are much more complex, in reality, than this simple overview. There are numerous interactions taking place among various immunological components such as cytokines, antigen presenting cells, as well as the T- and B-cells already discussed. This diversity may play a key role in the learning and adaptability of the immune system.

Additionally, the immune system is inherently distributed. Immune system cells can be found in a wide variety of places throughout the body: from the liver to the lymphnodes. This inherent distributedness suggests that there is no real central control of the immune system [96]. Each component has developed to function without direct supervision. Yet, this is not to imply that there is not cooperation within the immune system. One of the hallmarks (from a computational standpoint) of the immune system is the degree of cooperative (rather than competitive) problem solving exhibited in the system. There are interesting communication patterns demonstrated in cytokine networks as well as within the interactions of T- and B-cells [103, chap10, pp177-199]. Many of these capabilities are still being explored.

2.1.4 The Nature of Self

The discussion of T-cells and self recognition touched on in 2.1.1 was perhaps a bit simplified. As pointed out, one view of the immune system is that it defines

self from other. That is, the immune system is capable of defending us by clearly defining what should belong in the body and what should not. That this comes up at all, really, derives from the fact that the material the body is made up of (proteins, amino acids, etc.) is the same material that invaders are made up of as well. Given this, it seems logical that the immune system must have a way of knowing that it should not attack the body itself but that it should attack those invaders that do not belong in the body. Thus, the immune system must have a way of defining what is self and what is non-self. Naïvely, this makes perfect sense. However, this is one point of contention in the immunological world. Does the immune system really have this capability, or more importantly, some innate *a priori* knowledge of what is self and what is non-self? Recent theories from Matzinger [83] would say something like "The immune system does not define self and non-self, but it is capable of recognizing danger and non-danger." If this is more than just a relabeling of the terms (as Bersini warns us against [9]), then this might actually be telling us something. Otherwise it is fairly useless. On the other hand, we have Bersini, through the work of Varela, arguing that there is no such *a priori* knowledge built into the immune system that allows for the defining of self and non-self [9, 10, 11]. Rather, there is a concept of self-assertion—that the immune system, over time, emerges this picture of self through the interactions of the lymphocyte cells as well as the interaction of these cells with antigens. Antigens are then killed off only because they do not fit into this self-asserted scheme of the immune system. However, the immune system can develop zones of tolerance, which would, in fact, allow an antigen to exist in the system without

attacking it. It could then be argued that these zones of tolerance might fit nicely into this view of Danger/Non-Danger dichotomy. That is, the reason a zone of tolerance developed at all was that an antigen that appeared in that zone would cause no danger to the system.

2.1.5 Summary on Biological Immune Systems

This section has introduced some of the basic components or agents that play a role in immunological responses within biological immune systems. Obviously, this survey has been high-level and fairly incomplete. However, what this sketch does provide is a glimpse of a biological system which adapts and improves with experience. That is, it learns. The next section revisits the topic of artificial learning systems, and then the chapter concludes with a discussion of the use of these immunological principles as metaphors for the development of immune-inspired learning systems.

2.2 A Framework for Learning Algorithms

In section 1.1.2 we briefly touched on what is commonly meant by machine learning. In this section elaborate upon this further. We begin by introducing a very basic framework for examining learning algorithms. This allows us to provide some cohesion in our discussion of various immune-inspired learning algorithms to come. We then go on to discuss the classic major divisions of machine learning—supervised, unsupervised, and reinforcement—and provide examples of the use of

this framework for exemplars from these divisions. In this discussion, we also look at some of the pragmatics these different approaches require.

2.2.1 The MAD Framework for Learning

There are numerous ways of characterizing learning systems from the mechanisms they use to the philosophy they embody. In [86] the author chooses to focus on the way in which each learning technique manipulates the hypothesis space. This formalism allows for a fairly succinct means of comparison across various algorithms. Here we introduce a slightly different way of discussing learning. This is not introduced as a replacement for other methods of characterizing learning, but rather as a (fairly simplistic) framework to aid in our current discussion.

Any learning system can be discussed in terms of three key components: memory, adaptation, and decision-making. **Memory** involves how the system keeps track of past events or experiences and is used for the decision-making process on how to react to new experiences. The memory of a learning system is basically the system's abstracted view of the world—how it generalizes its experience. **Adaptation** includes the mechanisms that are used to modify this memory structure. It is where this world-view is adjusted when new experiences or feedback are encountered. **Decision-making** focuses on interaction with the environment. As the system undergoes an experiences, the system must decide what to do with that experience. Is it similar to something it has seen before (as recorded in the memory), if so, how does the memory system tell it to react? If

it is not, then is this something that it should try to remember? The memory system is adapted to encourage the learning of this new experience, and there can also be an adaptation of the memory to forget those aspects that are no longer relevant or to combine certain aspect of the memory to allow for abstraction or generalization.

To be truly learning, the memory system must never be just a look-up table of past experiences, but must exhibit the ability for abstraction or generalization so that sufficiently similar experiences are handled in a similar manner. Of course, the decision making should not be "should I remember this data item or this experience," but really should be "what action do I take now." The action could be to add something to the memory, perform some kind of action on the external environment, send out some kind of signal, label something as a given class, or anything else that might be appropriate for a given application or system or any combination of these.

Many learning algorithms are, in fact, two-phase algorithms. The first phase consists of training the system–that is the building of the memory system and the resultant decision making facilities. During this phase, the system is exposed to examples from the real world. Oftentimes, during training the system is exposed repeatedly to the same examples until some stopping criterion is reached (e.g., a fixed number of repetitions, the memory system has reached a certain structure, etc.) The second traditional phase for learning systems is the "test" or "production" phase. During this phase, the system could view its memory system as static, its adaptation mechanism as dormant, and the

system performs decision making based on incoming experiences and the memory system. For continuous learning systems (including most reinforcement learning systems), there is not really such a distinction between a "training" phase and a "testing" or "production" phase. The system is always adapting and learning as it interacts more with the environment. This seems to be closer to what biological systems would do: There is some encoding or training that occurs either through inheritance or initial embryogenesis, which would contain the rudiments of a memory structure and the basic understanding of the adaptive mechanisms necessary for learning, but most of the system learns as it goes with its interaction with the real-world providing the object lessons.[3] [104, 131]

Now that we have this memory, adaptation, decision-making framework roughly outlined, we can apply it to some of the more well-known learning algorithms. We start by looking at supervised learning, move onto unsupervised learning or clustering, and then examine reinforcement learning. We later use this framework to characterize immune-inspired learning algorithms, as well.

2.2.2 Supervised Learning

The field of machine learning can be divided into three broad categories: supervised, unsupervised, and reinforcement learning. With supervised learning, the ultimate goal is usually classification of a previously unseen data item based on the characteristics of the problem learned during the training phase of the

[3]This, it seems, is very much related to a "self-assertive" idea that denies most, if not all, *a priori* conditions to learning and instead insists that the immune system learns through interaction both within itself and with the outside world.

algorithm. During training, both a feature vector and the vector's classification is given. The algorithm uses both pieces of information to learn a view of the world that will allow correct classification of future instances. At times the known class of the training instance will be used to asses the quality of the algorithm's response; whereas, at other times this class information will be used for restructuring the memory representation of the system. In all cases of supervised learning, it is the combination of the feature vector and the class that help dictate the adaptation of the memory system.

Below, we look at two well-know supervised learning algorithms: decision trees and multi-layered perceptron (MLP) artificial neural networks using back propagation. Decision tree learning offers a symbolic representation of the decisions needed to classify any given instance in the problem space and uses the concept of information gain to develop this representation. MLP artificial neural networks are classic biologically-inspired learning systems that produce non-linear, subsymbolic representations of the decision space.

Decision Trees

One common learning technique is decision tree learning [101, 86, chap3, pp52-80]. This a means of approximating a target function which is represented as a decision tree. Each internal node in the decision tree is an attribute from the problem space, and each branch from a given internal node corresponds to a value the attribute can assume. The leaf nodes of the tree represent the values of the target attribute (the attribute that is being used for classification).

As may be obvious from this description, decision tree learning typically applies to problems whose feature vectors consist of discrete valued attributes. For example, suppose we are trying to classify whether a given day would be a good one to go to the cinema. Our feature vectors might consist of four attributes: the cinema's selection, the day's weather, the amount of money on hand, and if we have someone to accompany us to the cinema. These attributes can then take on a set of values. For example, maybe the cinema's selection attribute can take on the values of new or old; the weather attribute can take on the values wet, dry, or fair; the money attribute can take on the values film (enough for the film only), snacks (enough for the film and some snacks) or insufficient (not enough for even the film); and the companion attribute can take on the values yes or no. Figure 2.2.1 gives a possible decision tree using these features.[4] Then, the

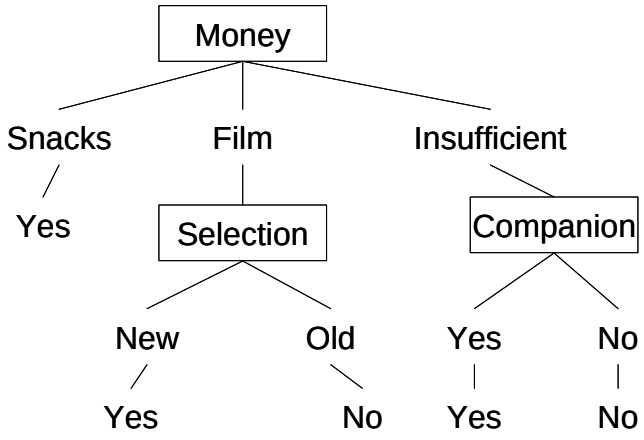

Figure 2.2.1: A decision tree for classifying days as good ones for going to the cinema

[4]As evidenced from this figure, in this hypothetical scenario, it was discovered that the "weather" variable was not needed to classify a day as a good one to go to the cinema.

decision tree could be used to classify a given day based on whether we should go to the cinema. For example, the feature vector <Selection=Old, Weather = Wet, Money = Film, Companion = No> would follow the center path of the tree and this instance would be classified as GoToCinema=no.

Decision tree learning, then, is the process of learning the correct structure of this tree, i.e., which attributes get tested at which level or along which path of the tree. This is typically achieved through finding the tree that correctly classifies the largest number of examples from a training data set. Once this tree is discovered, pruning, through various means, sometimes occurs to prevent the tree from overfitting the training data and, thus, not being able to generalize to previously unseen data items [38, 101]. Finally, the learned tree is used to classify instances of the problem space according to the target attribute.

Using our MAD framework, we can characterize decision tree learning algorithms as follows:

- Memory: The memory, or world view, of a decision tree is within the nodes and structure of the tree itself;

- Adaptation: Adaptation occurs in two places in most decision tree learning algorithms. In the initial building of the tree, the structure is constantly being adapted as different internal nodes are tried and rejected; in the pruning stage, this structure is again adapted based on various pruning rules.

- Decision-making: The decision-making step during training is a calculation of, based on what is already present in the tree (the memory structure), which attribute will provide the most information. This decision, in turn, guides the adaptation of the tree to possibly include this attribute as an internal node—or remove it at the pruning stage. During testing or classification, the decision-making step is simply an output of the value of the leaf node that is arrived at by following the path from the root to the leaf for the given instance.

Multi-Layered Perceptron Artificial Neural Networks

As mentioned in section 1.1.3, Artificial Neural Networks (ANN) have proven a successful and popular approach to machine learning tasks [12, 58, 86, chap4, pp81-127]. Perhaps the best-known learning algorithm from the field of ANN is the multi-layer perceptron network with the back-propagation algorithm used for weight adjustment. Figure 2.2.2 provides a visualization of a multi-layer, feed-forward network. In this figure F is a nonlinear activation function (oftentimes the logistic sigmoid) used to determine the output (y) of a given hidden layer node based on the input values, the weights of the connections between the input values and the hidden node, and a bias:

$$y_j = F(\sum_{i=1}^{n} w_{j,i}x_i + b_1). \tag{2.2.1}$$

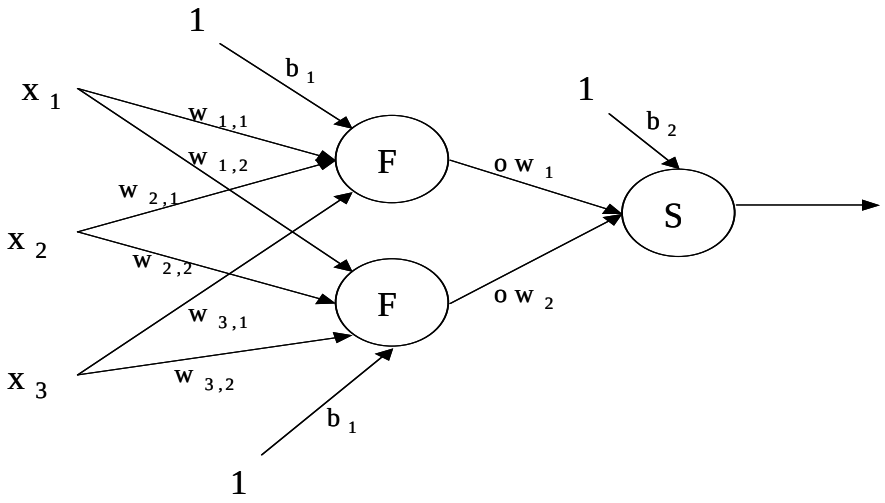

Figure 2.2.2: A feed-forward network: one hidden layer and one output

And S is often a linear function to determine the overall output of the system based on the output of the hidden layer, the weights connecting the hidden layer to the ouput, and another bias:

$$\text{output} = \sum_{j=1}^{\text{number of hidden nodes}} \text{ow}_j y_j + b_2. \tag{2.2.2}$$

The back-propagation algorithm is basically a gradient descent in the error space of the network's output. It serves as a method for assigning error across the weights of the network and guiding this parameter space toward the correct output function.

Multi-layer perceptrons with back-propagation learning, then, is an attempt to learn the best values of the weights in the network so that the output minimizes some error function. Training consists of multiple passes through the training data with the weights being adjusted based on the classification errors of this

data. Training often stops after the error rate has been reduced to a certain level. This prevents overtraining and allows for generalization, which in turn allows for accurate classification of previously unseen data items.

Using our MAD framework, we can characterize this neural network algorithm as follows:

- Memory: The memory structure of the neural network is contained within the weights of the network;
- Adaptation: Adaptation occurs by adjusting of weights; the adaptive pressure is motivated by the feedback from the environment through the back-propagation algorithm;
- Decision-making: Classification of a given experience is based on the value of the weights (dynamic memory) and the activation function of the nodes in the network. During training, this classification is then used as feedback into the system through the use of the back-propagation algorithm. The memory structure is adapted based on the errors of this decision.

2.2.3 Unsupervised Learning

The second major category of learning algorithms is unsupervised learning. These are often referred to as clustering algorithms because the primary goal of the algorithm is to discover interesting "clusters" of information. Unlike with supervised learning algorithms, unsupervised learners do not have access to the "correct" classification of a given data item. All an unsupervised learner is given

is the data itself. The task, then, becomes to learn or discover similarities within the training data set and to use this information to generalize about the problem space. During production or testing the algorithm makes a decision about how similar the new data item is to the learned material and this is then given to the user for interpretation.

As with our discussion of supervised learning, we walk through two examples of unsupervised learners with our MAD framework: one from a more classic learning stand point and one biologically inspired. For unsupervised learning, we explore the k-Means statistical based learning algorithm and the biologically inspired self-organizing maps.

k-Means

One classic unsupervised learning technique is k-Means [36, 99]. This algorithm is designed to cluster the given data points into k subsets. It proceeds by choosing k "centers" at random in the data and then assigning each point in the data set to one of these centers. Then the center is adjusted to truly be the center of the group of points that are assigned to it. This two stage process—assign a point to a center, adjust the center to reflect the points assigned to it—continues until the centers stop moving in two successive iterations. This algorithm is used for basic data analysis in the hope of discovering previously unknown relationships in the data. Upon termination, the data has been clustered into k groups, and members of a common group can be examined for commonalities.

k-Means learning is an attempt to find the optimal location for the centers that capture the inherent groupings within the data set. It does this by making a guess as to the best location and iteratively refining that guess.

Using our MAD framework, we can characterize the k-Means algorithm as follows:

- Memory: The centers of the clusters represent the memory structure of this algorithm;
- Adaptation: At each iteration, the position of the centers are adjusted based on the data points assigned to them;
- Decision-Making: The decision for a center to move is based on its surrounding data points (i.e., its experience); the decision for a data point to be assigned to a given cluster is based on its proximity to the center (memory point) of that cluster.

Self-Organizing Maps

One example of a biologically inspired unsupervised learning algorithm is Kohonen's Self-Organizing Maps (SOM) [75, 120]. SOM are inspired by the way neurons tend to cluster in the brain, and thus, SOM can be considered another type of Artificial Neural Network. As with our multi-layer perceptron, in SOM we have a collection of inputs that are connected to nodes in the network. However, this time there is no hidden layer but simply a single output layer. Typically this output layer is arranged as a 1 or 2 dimensional array with the assumption

that all the nodes are connected to each other. The goal in SOM learning is to modify the topology of the output nodes to represent the input space such that input values that are "close" together elicit responses from output nodes that are "close" together. This is achieved by iterating through the input space and updating the weights connecting the output layers based on a "neighborhood" effect. That is, those output nodes closest to the input item will be most strongly affected.

SOM learning is an attempt to find the weights, and thus topology, of the output layer that best reflects patterns inherent in the input space. Using our MAD framework, we can characterize the Kohonen's SOM algorithm as follows:

- Memory: The weights connecting the output layer's neurons which indicate their relative "position" to the other nodes in this layer;
- Adaptation: The adjustment of these weights as new input items are encountered;
- Decision-Making: During training, based on the input item and the current position of the nodes in the output layer, a decision is made on how to adjust the closest output nodes. During clustering, the regions of the output layer which respond to given inputs provide a means of analyzing the data.

2.2.4 Reinforcement Learning

The third major branch of machine learning is reinforcement learning. Reinforcement learning focuses on goal-driven learning in which the actions that

the learner takes has an impact on the external environment. However, this impact is possibly only dimly perceived. Unlike supervised learning which presents training examples and solutions as connected pairs, in reinforcement learning systems the learner is given a reward for certain actions. The goal of the learner is to maximize its reward. One of the fundamental issues in this reinforcement learning environment is determining which sequence of actions resulted in the greatest reward. That is, since the reward may be provided only after several actions made by the learner, the learner is taxed with the task of appropriately assigning credit to the given states it passed through when attaining the given reward.

One algorithm fundamental to reinforcement learning has been the Q learning algorithm [86, 109]. In this algorithm the goal is to learn the function Q which is the evaluation function for the learner based upon the current perceived state, previous actions, and the given reward. That is, learning Q is equivalent to learning the optimal policy the learner should follow. Using our MAD framework, we can characterize Q learning as follows:

- Memory: The state (or perceived state) of the learner.
- Action: A transition to a new state based upon the chosen action.
- Decision-Making: Updating of the value of a given state based upon the reward achieved when choosing the given action.

2.2.5 Some Final Remarks about the MAD Framework

We can continue this characterization of numerous machine learning systems. Table 2.2.1 places Support Vector Machines [88, 121], K-Nearest Neighbor [86, chap8, pp230-248], and Genetic Algorithms [85] within this framework.[5]

Table 2.2.1: MAD Learning Framework and Various Machine Learning Techniques

Technique	Memory	Adaptation	Decision Making
SVM	Collection of vectors	Adjustment of kernel widths (margins)	Classification based on vector decision boundaries
KNN	Training set	No real adaptation occurs	Classification based on majority vote
GAs	Population of individuals	Mutation and crossover among fittest individuals	Surviving individual is solution to problem

The application of this framework to other learning techniques could be continued *ad nauseam*. However, it does provide a common terminology for discussing learning algorithms. As we see in the following section, this allows us to make various immune-inspired algorithms more accessible in terms of this wider context. While we provide this simplified framework for just this purpose, we still want to acknowledge the legitimacy of other ways of examining learning algorithms. In fact, some of these ways may be more fruitful especially when the learning system is analyzed in terms of how it approaches the hypothesis space or

[5]Treating GAs as a learning system might be stretching the classic definition of learning a bit too far, unless you consider the Baldwin effect[6, 59] as learning. However, it does still fit nicely within this framework.

what learning biases are being used [46, 86]. Yet, this MAD framework gives us a good starting point for our discussion of immune learning.

One concept we have hinted at in this section is that biology has provided rich inspiration for the development of learning algorithms. In the next section we bring together the two main threads of this chapter: immunology and machine learning. We offer an overview of the development of some of the immune-inspired learning algorithms. We provide pointers to the biology that inspired this as well as the learning principles that are utilized.

2.3 Immune-Based Learning Systems

Now that we have seen some of the biology of the immune system, we turn our attention to artificial immune systems. Obviously, the biological discussion has been slanted in such a way to highlight some of the aspects that allow for the development of learning algorithms. We have seen that the immune system does have mechanisms that facilitate learning in some fashion. Through exposure to antigens, the immune system is capable of remembering the exposures, generalizing beyond one specific antigen, and reacting in a more appropriate manner when presented with the same or similar antigens. That is—hearkening back to our original definition of machine learning—it automatically improves with experience. Since this is evident in the biological system, how can we use some of what we see there to provide us a map or inspiration into building computational systems that learn?

Likewise, we have taken a brief look at some established learning paradigms. In this examination, we have introduced a framework for discussing learning algorithms. As we now turn our attention to immune-inspired learning, we appeal to this framework to provide a context to learning algorithms in general.

In this section we provide a fairly in-depth survey of immune-inspired learning algorithms. This survey starts by examining some of the theoretical ideas or schools of thought that have played key roles in the development of immune learning algorithms. We then move to several specific immune-inspired learning algorithms starting with population based algorithms and moving to network based models.

2.3.1 Immune Learning Theory

In this subsection we examine some of the key theoretical (for lack of a more succinct term) or philosophical discussions that have led to the ideas of many immune learning algorithms. While many of the works discussed here do not outline specific learning systems, they have often led to the design of such systems by other researchers.

In the late 1980s and early 1990s, JD Farmer and his colleagues started examining the potential learning capabilities of the immune system. In [39] the authors examine the similarities between the dynamics of Holland's classifier systems and the immune network model. And given these similarities, they go on to argue that "there may be certain universal approaches to the design of efficient

learning systems." This idea is made explicit in [40] when Farmer strives to provide a common methodology for discussing a wide variety of connectionist approaches, including immune networks. From the development of learning systems point of view, these early articles on the learning potential of the immune system have proven highly inspirational. Cornerstone concepts to the development of later-day immune-inspired learning algorithms, such as immunological forgetting, the parallel nature of the immune system's behavior, and the similarities between these simulated immune networks and artificial neural networks, can all be found in these early works by Farmer and his collaborators.

Varela, Bersini, and their colleagues have likewise often offered tantalizing pictures of the immune system as a learning adaptive system. In [122], the authors argue a self-assertive view of the immune system. They claim that shape-space exists in a void, but the interactions of immune cells and the formation of the immune network leads to the identity or personality of the immune system in this shape-space. This leads to an argument that the immune system is a cognitive system, and so a brief comparison between the immune system and the neural system is offered, with a strong case being made that the immune system has greater variance and dynamics. Varela, and through him Bersini, views antigenic interaction as a by-product of the immune system asserting its identity rather than its primary cognitive task. However, this may contrast some with the biological evidence. For example, the authors of [64] suggest that in the embryonic immune system, there seems to be evidence that the immune system does not fully develop

until antigenic presentation. Yet, the fact that the immune system exists outside of this presentation maybe enough to confirm Varela's point.

Bersini and Varela continued this self-assertive point of view throughout several subsequent papers. In [11], the authors discuss the similarities between immune network learning and reinforcement learning and emphasize the fact that the environment is not an instructor but a trigger for certain types of behavior. In [8], the authors argue that the novelty for engineering as inspired by the immune system, or at least strong inspiration in the sense of ANNs and GAs, will only come by looking not to the defensive by-products of the immune system (self/non-self) but to the self-assertion qualities. They differentiate immune systems and evolutionary/ecosystems by the fact that the immune systems' performances can be evaluated in a global fashion, i.e., immune networks have a collective function (thus, implying a sense of cooperation), and this is not seen in the evolutionary/ecosystems.

Rather than a GA approach as seen in classifier systems, there is an argument that if the collective performance of the system is what is of interest then the weakest and not the strongest member's abilities are more important. There is a need to maintain diversity and robustness and compensate for these weak members. They argue for the need of structural plasticity as well as parametric plasticity. (For reference sake, let us talk about an ANN. In an ANN, parametric plasticity would refer to the increasing or decreasing of the various weights in the system. Structural plasticity would involve adding or deleting neurons during

operation.) This paper examines the use of this concept of double plasticity and endogenous change in three application areas:

1. Artificial Neural Networks: demonstrated that not just the weights but the network structure can be "learned" or evolved.

2. Autonomous Agent Learning by Reinforcement: elaborated upon in [11]

3. Control of Chaos: discusses the idea of zones of effective control within a chaotic system (or around fixed point within a chaotic system).

Much of this is continued in [9], where Bersini argues that the immune system's key dynamic is along a self-assertion line rather than the traditional self/non-self dichotomy. He proposes that the adaptability of the system is to "satisfy endogenous constraints instead of responding to exogenous impacts" and that this will provide more benefits from the perspective of adaptive (learning) systems. There is a warning against the possible banal interpretation of danger theory as merely using a different naming scheme for the same concepts (e.g., using danger to mean non-self); however, he asserts there could be mileage in this theory if the view is that immune reaction is context-sensitive. That is, the immune system does not respond the same way every time to an antigen (which becomes a meaningless name), but rather the immune system's reaction is dependent upon the context in which a given interaction occurs. And this context is internally defined based on the immune system "knowing only itself."

While all of this exploration of the self-assertive nature of the immune system has provided for interesting discussion within the AIS community, very little has

been done to include this in a practical, engineering system. However, many of these concepts have proven critical for the exploration and application of immune system dynamics to learning systems.

A final "school of thought" should be mentioned in this subsection. It is impossible to deny the influence the work of Stephanie Forrest and her group at the University of New Mexico has had on the field of Artificial Immune Systems. While they have not often directly examined the role of learning in the immune system, much of their work has application to this field. Perhaps the most accessible summary of their work can be found in [42]. In this wide-ranging review, the authors argue that the immune system can be viewed as an information processing system. The paper discusses the roles negative selection, self/non-self, danger theory, gene libraries, and a host of other concepts have on the development of computational systems based on these immunological observations. It concludes with a discussion of the application of many of these ideas to network security.

2.3.2 Population Based AIS Algorithms

In this subsection we examine several population-based immune-inspired learning algorithms. By population-based we mean those algorithms based on the evolution of individual immune-like cells. This is in contrast with the explicit connections that are employed in the network-based models discussed in subsection 2.3.3.

Negative Selection

Perhaps the most influential idea to emerge from the field of artificial immune systems has been negative selection. In the biological immune system, negative selection refers to the maturation process of T-cells. T-cells develop in the thymus by being exposed to cells from the "self" (i.e., the body itself). Those T-cells which react to the "self" are not allowed to mature; whereas, those that do not react to the "self" after enough time are released into the body as mature T-cells. This idea has been exploited in the computational world most often in terms of security and anomaly detection.

The basic reasoning follows that if one can define what is meant by the "self" of a computer system, piece of software, network, etc. then one can generate a set of detectors that can recognize "non-self" or anomalies. This concept has been explored by Forrest and colleagues in a series of experiments from software change detection [33] to network security [61]. The heart of this work has been on the concept of being able to define the "self" of a system and then to examine the environment for anomalies [23, 26, 45, 43, 44, 63, 62]. Typically negative selection proceeds by developing a set of detectors which do not match the "self" of whatever system is being monitored. Then, if one of these detectors matches—or, has a high affinity for—something, an anomaly has occurred.

While this concept has been greatly exploited in the AIS field, it has often been abused, as well. As Kim points out in her thesis, "...the job of a negative selection stage should be restricted to tackle a more modest task that is closer

to the role of negative selection of human immune system. That is simply filtering the harmful antibodies rather than generating competent ones..."[72]. Too often, however, within the AIS field we see negative selection being used to overcome a deficiency in data. That is, the negative selection algorithm is used to generate counter examples to whatever is being learned to avoid having to collect large amounts of "anomalous" data. When this application of negative selection occurs, the algorithm is basically reduced to a 2-class classification algorithm—classifying between "self" and "non-self". For examples of this misuse of the negative selection algorithm, see [48, 94, 112], to name a few. By "misuse" we mean in the same sense as Kim pointed out in her thesis. Many times the negative selection algorithm is used for data generation when its function in nature is limited more toward data filtering. This abuse at times leads to complex algorithms that attempt to apply negative selection to inappropriate situations. One interesting example from this type of approach, however, can be found in [24]. While it is still, ostensibly, dealing with anomaly detection, this algorithm tries to integrate two different types of T-cells as well as B-cells. The authors claim better performance than simple negative selection algorithms. Although the negative selection algorithm is still part of this work, it is interesting from the perspective of adding a multi-cellular approach which has had limited exploration within the AIS field. However, like most negative selection based learning algorithms, it is limited in its need for a clearly defined concept of "self" before starting. As with many anomaly detection algorithms in AIS, this seems to be nothing but a 2-class

classification algorithm. The real novelty to this work is the use of multiple types of immune cells within one algorithm.

To apply our learning framework to the negative selection algorithm, then:

- Memory: A collection of detector cells are employed to represent the "non-self" space;
- Adaptation: The collection of immature detectors is exposed to the "self" space for a certain amount of time. Those immature detectors which do not react to the "self" cells are allowed to become mature detectors;
- Decision-making: The primary decision is made based on the mature detectors' reactions to incoming antigens. If the mature detectors react strongly enough, then the antigen is labeled "non-self" and a possible secondary reaction occurs. During the maturation phase, the decision is based on the immature detectors' reactions to the "self" space. Those immature detectors which react strongly enough to the "self" are removed from the system rather than being allowed to mature.

Gene Libraries

Another early population-based exploration was the study of the role of gene libraries on the development of effective antibodies. A gene library is a collection of antibody "building blocks" or patterns from which individual antibodies can be built. That is, a gene library consists of a group of pattern (or gene) fragments which can be combined to form all of the antibodies in a system. Since antibodies

are one of the primary pattern recognition cells within the immune system, the way in which they develop can provide insight for solving other pattern recognition tasks. Hightower examined this issue in a series of papers. In [60] Hightower introduced the concept of immune libraries and distinguished between the phenotype of the immune system (the actual antibodies) and the genotype (the make-up of the gene libraries). This paper demonstrated an interesting result suggesting that subjecting just the phenotype to fitness evaluation, and thus selective pressure, can indeed cause an improvement in the genotype. That is, the gene libraries (or the potential antibody repertoire) of the evolved individuals were moved toward ideal antigen coverage only through subjecting the actual antibodies (or the expressed antibody repertoire) to the genetic pressure inherent in a GA. Hightower followed this up in [59] to incorporate learning by allowing an "antibody" to make a certain number of guesses at the correct solution, thus performing a local search and simulating somatic hypermutation. It was shown that this learning, while not written directly back to the genes, still improved the fitness of successive generations through the Baldwin effect: that is, the individuals that learned the best survived to pass on their original genes to successive generations with the idea that these genes contained good templates, so to say, for improved performance. It also indicated that with a finite learning task, there can be such a thing as too much learning which reduces the evolutionary pressure by hiding the genetic defects of poorer individuals who were given enough time to "master" or learn the task more perfectly. To place this work in context of our framework introduced in 2.2.1:

- Memory: The memory of this system is the antibodies which are developed from the combination of patterns present in the gene library;
- Adaptation: Adaptation occurs through somatic hypermutation and survival of the fittest antibody. This, in turn, rewards those individuals whose gene libraries contained patterns appropriate for developing this best individual;
- Decision-making: The decision is based on a fitness or affinity measure used to choose which antibodies are allowed to mutate and thus "learn" the task at hand.

The exploration of the use of gene libraries was continued from this work by Hightower, most notably in the work of Oprea [98]. While the focus in this more recent work is no longer on learning, the ideas play a role in other learning algorithms, for example in Kim's DynamicCS algorithm discussed later in this section [72].

Clonal Selection

Along with negative selection, discussed earlier, the immunological concept of clonal selection has been one of the most popularly exploited ideas for the building of artificial immune systems. Biologically, the concept is fairly simple: the antibody (or cell) which exhibits the greatest affinity for an antigen is selected for propagation (cloning) and mutation [16]. Clonal selection is, in fact, very similar to the evolutionary idea of survival of the fittest that have inspired

many evolutionary approaches to computing (e.g., genetic algorithms, genetic programming, etc.).

The idea of clonal selection has been most succinctly captured in de Castro's CLONALG algorithm [32]. We will discuss this algorithm in much more detail in chapter 5. However, to give a high-level overview of the algorithm here, CLONALG's primary purpose is to develop a set of memory cells which represent the data to be learned. This is achieved through a series of presentations of the training data to the algorithm. The artificial cells of the system are allowed to react to this data with those with the highest affinity being allowed to clone and mutate. Mutation is done (as seen in our previous discussion of gene libraries) in an affinity proportional matter. That is, those cells with greater affinity for the antigen mutate less than those with lesser affinity. Both a learning (pattern recognition) and optimization version of this algorithm was developed. We can cast CLONALG in terms of our framework as follows:

- Memory: A group of evolved memory cells which represent patterns to be recognized;
- Adaptation: Each antigen is presented to every antibody. The best (highest affinity) antibodies get a chance to clone, undergo somatic hypermutation, and potentially replace an existing memory cell;
- Decision-making: During training, the basic decision-making process is based on the current memory cells' and antibodies' reactions to the antigenic pattern (i.e., clonal selection). After training the decision of how to

identify/recognize an incoming pattern is based on the affinity of the memory cells (in particular the one memory cell with the highest affinity) to the given test pattern.

While there are still a number of issues with this algorithm and its usefulness for general pattern recognition tasks, it often provides a good base-line immune algorithm to build other algorithms upon.

A good example of an AIS that employs all of the factors discussed in this section (i.e., negative selection, gene libraries, and clonal selection) is the work of Jungwon Kim [72]. Kim documented the evolution of a clonal selection based algorithm and examined the impact of adding various immunological components had on the overall algorithm. Her primary interest was ultimately intrusion detection; however, she cast most of her work in terms of a two-class classification problem in order to capture this (now) classic concept of "normal" and "anomalous" or "self" and "non-self." She began by evaluating her algorithm, StatiCS, as a two-class classification problem. Obviously, this was just to get a handle on the behavior of the algorithm and not to propose a new classification/supervised learning algorithm, per se. Still, we come back to the idea of treating this arbitrary division of self/non-self as a classification issue. This requires, as all negative selection based algorithms do, a clear definition of self, and thus supervision (in a broad sense of the word).

In DynamicCS the overall cell population size was fixed. This algorithm incorporates the idea of cell evolution from immature to mature to memory cells

based on time. It is still focusing on self/non-self and introduces the concept of costimulation from Hofmeyer [62]. Kim claims that "as long as immature detectors have an opportunity to experience various antigen distributions for a sufficient period, which is defined by the tolerisation period, the FP [false positives] can be dramatically reduced to an almost perfect near-zero rate." However, this may not be completely accurate. The fact that both the true positive and false positive rates decrease does not seem to be that much of an accomplishment. Basically, the system just does not learn to make a positive decision at all and instead will simply guess the negative class more often. This is the classic problem of all negative-selection based schemes: one has to be able to define what self means *a priori*. This does not seem to be the way the real immune system works. There is nothing in the system that says "I am self," but rather this knowledge is gained over time. Maybe we can have some embryonic notion of self which grows and evolves—essentially what Kim wants here in DynamicCS, but we cannot treat the system as having a complete knowledge of self, no matter how much we limit the impact of negative selection on the overall workings of the algorithm.

In extended DynamicCS the idea of dying memory cells is incorporated into the overall scheme. This becomes basically a form of supervised learning, with a bit of "cheating" involved, since the death of a memory cell is triggered by its detection of "self" which is confirmed by a human operator. Since, what Kim is really concerned with is a security system rather than a learning system, this might be acceptable, but from a machine learning standpoint, it loses much of its automation and ability to really learn on its own.

What is interesting about all of this work by Kim is the integration of some basic AIS techniques that have been used individually, but have not, necessarily, been coupled together. Namely, she integrates negative selection (and overcomes a few of the problems with this by realizing the appropriate place for this algorithm), clonal selection, and gene libraries.

2.3.3 Network Based Algorithms

Distinct from the AIS based solely on populations of independent cells has been those based on the Immune Network theory of Jerne [69]. This network concept proposes that there are interactions among B-cells within the body such that a B-cell is affected not only by an invading antigen but also by other B-cells nearby. This interaction provides both stimulation and suppression of an individual cell's response in any given situation, and, thus, a network of interactions is formed.

Early work in adopting this network metaphor to Artificial Immune Systems include works by Hunt [66] and Aisu [3]. Hunt's algorithm applies the network idea to the field of case based reasoning. It utilizes clonal selection and mutation, and it relies on the immune network dynamics for forgetting of useless cases. There is an emphasis placed on the self-organizing nature of the immune network as useful for growing appropriate memory structure in a Case-Based Reasoning (CBR) system which allows classification to occur. However, it does requires some manual manipulation at the classification stage. Aisu's work attempts to combine the network model with anomaly (self/non-self) detection. It still requires a basic

idea of self to be defined *a priori*. However, the authors claim that anomaly detection is just one example application and that their methodology would be more applicable to other learning domains.

This work was joined by that of Nikolaev [105] a few years later. In this, the authors present an application of network ideas to the k-state DFA problem. The paper emphasizes the importance of the self-organizing nature of the network for learning; Claims are made that unlike GAs, the immune learning algorithm presented would have a greater capacity for escaping from local optima. An analogy between the behavior of cells in the immune network and the behavior of particles thrown at a hill is made, and the authors use this analogy to define kinetic and potential energy models for the dynamics of the immune cells. This comparison to these physics based models has not been fully explored in later AIS literature. Also of note in this paper is the idea that the learning algorithm goes through cycles of three search phases "from oscillatory through asymptotic to chaotic alternate continuously, and the system self-regulates." This should be observable in most network-based algorithms and again has not been addressed in current network-based investigations.

By far the most prolific investigator of the immune network theory for learning has been Jon Timmis. In [116], Timmis introduces an immune network algorithm for unsupervised learning. This algorithm not only utilizes immune network dynamics, but also incorporates clonal selection and mutation. In terms of our framework, this initial algorithm could be characterized as using:

- Memory: The immune network which is dynamically adjusted through a decreasing network affinity threshold maintains the memory of the system;
- Adaptation: Adaptation occurs through clonal selection coupled with mutation; removing the weakest 5% of cells (where weakest is defined through the network interactions);
- Decision: Network reaction to antigenic presentation causes restructuring of memory structure (possibly), but just the reaction itself indicates a decision made by the system with regards to the incoming data item (experience).

This is an unsupervised learning technique and so requires manual inspection of results to ascertain quality. The network structure and its reaction to a "test" data item indicates its similarity to certain regions of the memory system (network). In some ways, then, it is a content-addressable memory system: different reactions by the network to test data items indicate a kind of classification of that data item or a different region of memory being accessed.

This work was followed by [117] which introduces the concept of shape-space and greater immunological memory abstraction to the previous work. Adaptation is now encouraged through competition for resources which places even more pressure on the cells of the network to evolve toward good representations of the data set; however, stimulation (and, consequently resource allocation) is still based on intra-network interactions along with responses to antigens. The memory system is seeded with the "antigen" set.

- Memory: Again, the immune network (pre-seeded with a portion of the antigen set) maintains the memory of the system;
- Adaptation: Adaptation occurs through clonal selection with mutation, stimulation through network interactions and antigenic presentation, and competition for resources with those cells that have no resources being culled from the memory system;
- Decision: Same as in previous work; however, with the concept of shape-spaces introduced we now have a greater ability to generalize from previous experiences than before with the highly populated networks.

This was then followed by [73] which began as a reimplementation of the algorithm in [117]. However, a propensity for immunological forgetting or of movement toward the strongest, or most represented, pattern in the data set rather than the ability to maintain a continuous model of the data was discovered. Also, this implementation exhibited extreme sensitivity to the size of the initial seeded network. While there is an argument that this drift in the memory toward the strongest data pattern is beneficial, there does seem to be a decrease in the actual learning capability of the algorithm. Basically, the memory representation is no longer indicative of the entire world-view but only that which was most represented. However, this may be desirable.

Inspired by this work, Knight introduced a three-layered learning system with a free-antibody layer, a B-cell layer, and a memory cell layer [74].[6] In Knight's algorithm, there is no longer a need for the seeding of the system; the system can evolve its own representation of the data. However, the initial training instances will be co-opted as cells in the system if need be. The heart of the representation is really the B-cell layer, which produces free-antibodies as well as memory cells. Ultimately, the data analysis is done using the resultant memory-cell layer, which, despite Knight's statements, actually does play some role in the learning.

- Memory: 3-layers maintain the system's representation of the world: free-antibodies, B-cells, and memory cells; ultimately, from a user's perspective, it is the memory cells which allow for data understanding/clustering, but this layer only develops through the interaction of the other layers in the system;

- Adaptation: There are several mechanisms of adaptation in use in this system. Basically, there is a chain reaction in which the response in the free-antibody layer causes a response in the B-cell layer (which feeds back to the free-antibody layer). The B-cell response, in turn, causes a response (or promotion to) the memory cell layer. There is also still the concept of binding via affinity, clonal selection, and somatic mutation. Affinity based binding occurs in the free-antibody layer (this can still be viewed as basically clonal selection). In the B-cell layer, clonal selection occurs through

[6] This work seems similar to DynamicCS [72] with a naïve layer, a mature layer, and a memory cell layer.

binding, and then stimulation is based on the free-antibodies gathered by this B-cell over time. (It is not completely clear how exactly the B-cell gets these free-antibodies.) This B-cell then undergoes directed mutation before being promoted (potentially) to the memory cell layer.[7] A potential memory cell is only incorporated into the memory cell pool if it is "far" enough (in the shape-space) away from existing memory cells. Finally, cell death (population control) is based on the interactions of the cells within the system. If a cell has not undergone any interaction throughout an iteration of antigenic presentation, then it is culled from the system. Also, free-antibodies that attach to an antigen are removed from the free-antibody pool.

- Decision: Since the final representation of the data is really only important at the memory cell layer for this application, it is here where the ultimate decision is made. There is a decision on how to link the cells in the memory cell layer based on their distance from one another, which represents clusters within the original training set.

As a cluster-finding tool, this system seems to be valuable, but there is not really a sense of a reaction to a previously unseen data item (i.e., no real sense of presenting a "test" set to the system once it has been trained). This work is also valuable

[7]It appears that the order of presentation in the B-cell pool will be highly significant. Since antigenic presentation at the B-cell layer stops once the first B-cell is found below the affinity threshold, there is no guarantee that this is the "best" or "closest" B-cell to the given antigen within the B-cell layer at that time. It is unclear if antigenic presentation is random within the B-cell layer, or if the order is predetermined.

in that it explores the creation of a stable immune memory structure that can be exploited in other immune-inspired algorithms.

Following on from his collaboration with Timmis in [117], Mark Neal has explored the use of immune networks for continuous learning and stability. Beginning in [91] and culminating in [92], Neal has produced a network-based continuous learning algorithm. This work is a development out of the algorithms presented in [113, 117]; however, much of the "instability" discovered in [73] has been removed. Neal has managed to achieve continuous learning without over-fitting the data. This is done, primarily, by removing any central control (such as resource allocation) in the previous algorithms as well as by removing the stochastic nature of these previous algorithms. With this, Neal was able to use the immune network theory to provide a "meta-stable" memory system.

Following within this vein of revisiting the network algorithms proposed in [113], Nasraoui and colleagues have examined the applicability of fuzzy computing techniques for immune networks as well as simplifications that allow for greater scalability [90, 89]. In these works, the authors lodge a complaint against current network-based unsupervised learning schemes as being unscalable due to the representation and growth of the networks. They have developed a method to reduce the memory and computational needs of network-based systems by combining other clustering techniques (namely k-means) with the AIS methods.

Developed at roughly the same time as the work in [117], de Castro introduced a different network inspired learning tool named aiNet [31] While it offers a

somewhat confusing network model, it does incorporate the memory cell concept similar to [32]. Within our framework we have:

- Memory: Memory is maintained within the network connections or memory cells; however, it appears that the memory is found more in the memory cells than within the connections themselves. According to the authors' discussion of the memory structure: "In the aiNet case, the nodes work as internal images of ensembles of patterns (thus representing the acquired knowledge), and the connection strengths describe the similarities among these ensembles. On the other hand, in the neural network case, the nodes are processing elements while the connection strengths may represent the knowledge."

- Adaptation: It is here that the network idea is more fully realized with suppression (and thus death) of individual cells being dependent (to some degree) on the similarity between a given cell and its surrounding cells. There is also the concepts of clonal selection and affinity maturation involved here with the fairly typical affinity proportional mutation.

- Decision: During training, the basic decision-making process is based on the current memory cells' reactions to the antigenic pattern (i.e., clonal selection); however, after training, there appears little automated decision making; rather analysis of the overall memory structure is done, perhaps by experts, to use that structure as useful analysis of the data-set in question.

The developed network is often combined with known statistical tools to aid in interpretation, and it showed some success on a handful (three) benchmark, unsupervised learning problems.

2.3.4 Summary of Immune-Based Learning Systems

This section examined many of the key ideas that have been employed for building immune-inspired learning systems. We began our discussion with a look at some of the more philosophical views of the role learning and cognition plays in biological immune systems. We then took a more practical look at population-based and network-based learning algorithms. We saw that there have been numerous immunological components used in the population-based algorithms including negative selection, gene libraries, and clonal selection. Many of these ideas are also carried over to the network-based algorithms which have the added component of simulating interactions among the artificial immune cells in order to create a network of reactions.

2.4 Summary

We began this chapter by taking a high-level overview of biological immune systems. We used this survey to motivate the idea that the biological system does indeed exhibit the capability for learning. This capability provides hints for building learning systems based on the mechanisms seen in the biological system. We then looked in more depth at what is meant by learning and learning

systems. We proposed that all learning systems can be discussed in terms of three components: memory, adaptation, and decision-making. This proposal allowed us to discuss various learning systems with a set of common terminology. This was followed by a discussion of immune-inspired learning systems. Several of the key artificial immune algorithms were discussed and placed within context of our learning framework.

In the next chapter we provide a detailed examination of one specific immune-inspired learning algorithm: the Artificial Immune Recognition System. This algorithm provides a clear example of the exploitation of immunological metaphors for supervised learning. We discuss how the incorporation of additional immunological components into this initial algorithm provides an overall simplification of the algorithm while retaining its classification properties.

Chapter 3

Artificial Immune Recognition System

This chapter looks in-depth at one immune-inspired algorithm that we have developed for supervised learning, named the Artificial Immune Recognition System (AIRS).[1] AIRS (Artificial Immune Recognition System) is a novel immune inspired supervised learning algorithm [124]. As with many biologically-inspired algorithms, the inspiration for AIRS initially developed at the crossroads of standard computing and biology by identifying a computational problem that had yet to be addressed by immune-inspired algorithms. This chapter is offered as an in-depth introduction to one immune-inspired algorithm. It also illustrates the way further study of the original biological system can provide beneficial insights into the continual development of efficient bio-inspired systems.

[1] This chapter is a slightly modified version of [130].

Initially, motivation for the development of AIRS came from our identification of the fact that there was a significant lack of research that explored the use of the immune system metaphor for supervised learning; indeed, the only work identified was that of [19]. However, as pointed out in section 2.3, within the AIS community there had been a number of investigations on exploiting immune mechanisms for unsupervised learning (that is, where the class of data is unknown *a priori*) [116, 117, 30]. As previously mentioned, work in [29] examined the role of the clonal selection process within the immune system [16] and went on to develop an unsupervised learning algorithm known as CLONALG. This work was extended by employing the metaphor of the immune network theory [69] and then applied to data clustering. This led to the development of the aiNet algorithm [30]. Experimentation with the aiNet algorithm revealed that evolved artificial immune networks, when combined with traditional statistical analysis tools, were very effective at extracting interesting and useful clusters from data sets. aiNet was further extended to multimodal optimization tasks [27]. Recall that other work in [116] also utilized the immune network theory metaphor for unsupervised learning, and then augmented the work with the development of a resource limited artificial immune network [117], which reported good benchmark results for cluster extraction and exploration with artificial immune networks. The work in [117] was of particular relevance to [124] and the developments presented in this chapter. Specifically, the ideas of artificial recognition balls and resource limitation from [117] and long-lived memory cells from [30] were most influential. However, while these population control mechanisms and data representation concepts were

borrowed from this work on immune networks, it should be stressed that AIRS is in **no way** an immune **network** model of computation.

This chapter begins in section 3.1 by examining some of the immune metaphors which have been employed within AIRS—most of which will echo what was discussed in chapter 2. Then follows a presentation of the initial algorithm as formulated in [124, 128]. This is followed by a discussion of the results obtained with this initial algorithm and some observations for simplifying the algorithm. As a contribution original to this thesis, we then present refinements to AIRS that aid in this simplicity which is followed by results indicating the benefits of these refinements. We conclude this chapter with a comparative look at the initial version of AIRS and our refinements.

3.1 AIRS: Immune Principles Employed

A little time should be taken to draw attention to the most relevant aspects of immunology that have been utilized as inspiration for this work. Here we highlight the more salient points of the immune system mentioned in section 2.1. Throughout a person's lifetime, the body is exposed to a huge variety of pathogenic (potentially harmful) material. The immune system contains lymphocyte cells known as B- and T-cells, each of which has a unique type of molecular receptor (location in a shape space). Receptors in this shape space allow for the binding of the pathogenic material (antigens), with higher affinity (complementarity) between the receptor and antigen indicating a stronger bind. Work in [28] adopted

the term shape-space to describe the shape of the data being used, and defined a number of affinity measures, such as Euclidean distance, which can be used to determine the interaction between elements in the AIS. Within AIRS (and most AIS techniques) the idea of antigen/antibody binding is employed and is known as antigenic presentation. When dealing with learning algorithms, this is used to implement the idea of matching between training data (antigens) and potential solutions (B-cells). Work in [117] employed the idea of an artificial recognition ball (ARB), which was inspired by work in [39] describing antigenic interaction within an immune network. Simply put, an ARB can be thought to represent a number of identical B-cells and is a mechanism employed to reduce duplication and dictate survival within the population. Once the affinity between a B-cell and an antigen has been determined, the B-cell involved transforms into a plasma cell and experiences clonal expansion. During the process of clonal expansion, the B-cell undergoes rapid proliferation (cloning) in proportion to how well it matches the antigen. This response is antigen specific. These clones then go through affinity maturation, where some undertake somatic hypermutation and eventually will go through a selection process through which a given cell may become a memory cell. These memory cells are retained to allow for a faster response to the same, or similar, antigen should the host become re-infected This faster response rate is known as the secondary immune response. Within AIRS, the idea of clonal expansion and affinity maturation are employed to encourage the generation of potential memory cells. These memory cells are later used for classification. Drawing on work from [117], AIRS utilized the idea of a stimulation

level for an ARB, which, again, was derived from the equations for an immune network described in [39]. Although AIRS was inspired by this work on immune networks, the development of the classifier led to the abandoning of the network principles in favor of a simple population-based model. In AIRS, ARBs experience a form of clonal expansion after being presented with training data (analogous to antigens); details of this process are provided in section 3.2. However, AIRS did not take into account the principle of affinity proportional mutation. When new ARBs were created, they were subjected to a process of random mutation with a certain probability and were then incorporated into the memory set of cells should their affinity have met certain criteria. Within the AIRS system, ARBs competed for survival based on the idea of a resource limited system [117]. A predefined number of resources existed, for which ARBs competed based on their stimulation level: the higher the stimulation value of an ARB the more resources it could claim. ARBs that could not successfully compete for resources were removed from the system. The term metadynamics of the immune system refers to the constant changing of the B-cell population through cell proliferation and death. This was present in AIRS with the continual production and removal of ARBs from the population.

3.2 The AIRS Algorithm

The previous section outlined the metaphors that were employed in the development of AIRS. This section now presents the actual algorithm and

discusses the results obtained from experimentation. A more detailed description of the algorithm can be found in [124], much of which is reproduced in Appendix A. Within AIRS, each element (ARB) corresponds to a vector of n dimensions and a class to which the data belongs. Additionally, each ARB has an associated stimulation level as defined in equation 3.2.1, where x is feature vector of the ARB, S^x is the stimulation of ARB x, y is the training antigen, and affinity(x, y), in the current implementation, is a function that calculates the Euclidean distance:[2]

$$S^x = \begin{cases} 1 - \text{affinity}(x, y) & : \quad \text{if class of } x \equiv \text{class of } y \\ \text{affinity}(x, y) & : \quad \text{otherwise} \end{cases} \tag{3.2.1}$$

Notionally, AIRS has four stages to learning: initialization, memory cell identification, resource competition, and finally refinement of established memory cells. AIRS is a one-shot learning algorithm; therefore, the process described below is run for each antigenic pattern, one at a time. Each of these processes will be outlined with the algorithm summarized below. Initialization of the system includes data pre-processing (normalization) and seeding of the system with randomly chosen data vectors from the training set. Assuming a normalized input training data set (antigens), data from that set are randomly selected to form the initial ARB population P and memory cells M. Prior to this selection, an affinity threshold is calculated; this threshold for the current implementation is the average Euclidean distance between each item in the training data set. This

[2]Euclidean distance was chosen as a simple starting point for these investigations. In theory, the AIRS algorithm should exhibit similar learning characteristics regardless of the affinity metric used. Experiments on the impact of this metric are currently under investigation [56].

is then used to control the quality of the memory cells maintained as classifier cells in the system. AIRS maintains a population of memory cells M for each class of antigen, which, upon termination of the algorithm, should provide a generalized representation for each class of antigenic pattern. The first stage of the algorithm is to determine the affinity of memory cells to each antigen of that class. Then the highest affinity cells are selected for cloning to produce a set of ARBs (which will ultimately be used to create an established memory set). The number of clones that are produced is in proportion to the antigenic affinity, i.e., how well they match; the ARBs also undergo a random mutation to introduce diversification. The next stage is to identify the strongest ARBs based on affinity to the training instance; these will be used to create the established memory set used for classification. This is achieved via a resource allocation mechanism, taken from [117], where ARBs are allocated a number of resources based on their normalized stimulation levels. At this point, it is worth noting that the stimulation level of an ARB is calculated not only from the antigenic match, but also the class of the ARB. This, in effect, provides reinforcement for ARBs that are of the same class as the antigenic pattern being learned and that match the antigenic pattern well, in addition to providing reinforcement for those that do not fall into that class and do not match the pattern well. Once the stimulation of an ARB has been calculated, the ARB is allowed to produce clones (which undergo mutation). The termination condition is then tested to discover if the ARBs are stimulated enough for training to cease on this antigenic pattern. This is defined by taking the average stimulation for the ARBs of each class, and if each of these averages

falls above a pre-defined threshold, training ceases for that pattern. This ARB production is repeated until the stopping criteria are met. Once the criteria have been met, then the candidate memory cell can be selected. A candidate memory cell is selected from the set of ARBs based on its stimulation level and class, with the most stimulated ARB of the same class as the antigen being selected as the candidate. If this candidate cell has a higher stimulation than any memory cell for that class in the established memory set **M**, then it is added to **M**. Additionally, if the affinity of this candidate memory cell with the previous best memory cell is below the affinity threshold, then this established memory cell is removed from the population and replaced by the newly evolved memory cell, thus achieving population control. This process is then repeated for all antigenic patterns. Once learning has completed, the set of established memory cells **M** can be used for classification. The algorithm is presented below, in terms of immune processes employed.

1. *Initialization*: Create a set of cells called the memory pool (**M**) and the ARB pool (**P**) from randomly selected training data.

2. *Antigenic Presentation*: for each antigenic pattern do:

   (a) *Clonal Expansion*: For each element of **M** determine their affinity to the antigenic pattern, which resides in the same class. Select highest affinity memory cell (*mc*) and clone *mc* in proportion to its antigenic affinity to add to the set of ARBs (**P**)

(b) *Affinity Maturation*: Mutate each ARB descendant of this highest affinity *mc*. Place each mutated ARB into **P**.

(c) *Metadynamics of ARBs*: Process each ARB through the resource allocation mechanism. This will result in some ARB death, and ultimately controls the population. Calculate the average stimulation for each ARB, and check for termination condition.

(d) *Clonal Expansion and Affinity Maturation*: Clone and mutate a randomly selected subset of the ARBs left in P based in proportion to their stimulation level.

(e) *Cycle*: While the average stimulation value of each ARB class group is less than a given stimulation threshold repeat from step 2.c.

(f) *Metadynamics of Memory Cells*: Select the highest affinity ARB of the same class as the antigen from the last antigenic interaction. If the affinity of this ARB with the antigenic pattern is better than that of the previously identified best memory cell *mc* then add the candidate (mc-candidate) to memory set **M**. Additionally, if the affinity of *mc* and mc-candidate is below the affinity threshold, then remove *mc* from **M**.

3. *Cycle*. Repeat step 2 until all antigenic patterns have been presented.

4. *Classify*. Once these 3 steps have been undertaken, the memory set **M** can be used to classify data items. Classification is performed in a k-Nearest

Neighbor fashion with a vote being made among the k closest memory cells to the given data item being classified.

This algorithm is presented more formally in Appendix A. Section 3.4 will employ the formalisms presented there when discussing the changes to this initial algorithm. To use our framework presented in section 2.2, we can characterize AIRS as follows

- Memory: The memory of the AIRS algorithm is in the pool of memory cells developed through exposure to the training data (experiences);
- Adaptation: The adaptation occurs primarily in the ARB pool. With each new experience, AIRS evolves a candidate memory cell in reaction to this experience. If this memory cell is of sufficient quality, then the memory structure is adapted to include it.
- Decision-making: The initial decision is which memory cell is most like the incoming training antigen. This cell is used as a progenitor for a pool of evolving cells. During classification, the primary classification decision is based on the k most similar memory cells to the data item being classified.

3.3 AIRS: Initial Results and Discussion

AIRS was tested on a number of benchmark data sets in order to assess the classification performance. This subsection will briefly highlight those results and discuss potential improvements for the algorithm; however, more details can be

found in [126]. Once a set of memory cells has been developed, the resultant cells can be used for classification. This is done through a k-nearest neighbor approach. Experiments were undertaken using a simple linearly separable data set, where classification accuracy of 98% was achieved using a k-value of 3. This seemed to bode well, and further experiments were undertaken using the Fisher Iris data set, Pima Diabetes data, Ionosphere data, and the Sonar data set, all obtained from the repository at the University of California at Irvine [13]. Table 3.3.1 shows the performance of AIRS on these data sets when compared with other popular classifiers [34] and [35], and a discussion of these comparative results can be found in [126].[3]

These results were obtained from averaging multiple runs of AIRS, typically consisting of three or more runs and five-way, or greater, cross validation. More specifically, for the Iris data set a five-fold cross validation scheme was employed with each result representing an average of three runs across these five divisions. To remain comparable to other experiments reported in the literature, the division between training and test sets of the Ionosphere data set as detailed in [13] was maintained. However, the results reported here still represent an average of three runs. For the Diabetes data set a ten-fold cross validation scheme was used, again with each of the 10 testing sets being disjoint from the others, and results were averaged over three runs across these data sets. Finally, the Sonar data set utilized the thirteen-way cross validation suggested in the literature [13] and was averaged

[3]For the Diabetes data set, 11 others reported with lower scores, including Bayes, Kohonen, kNN, ID3...

Table 3.3.1: Comparison of AIRS and Other Classifiers' Classification Results on Benchmark Data

	Iris		Ionosphere		Diabetes		Sonar	
1	Grobian (rough)	100%	3-NN + simplex	98.7%	Logdisc	77.7%	TAP MFT Bayes	92.3%
2	SSV	98.0%	3-NN	96.7%	IncNet	77.6%	Naïve MFT Bayes	90.4%
3	C-MLP 2LN	98.0%	IB3	96.7%	DIPOL92	77.6%	SVM	90.4%
4	PVM 2 rules	98.0%	MLP + BP	96.0%	Lin. Dis. Ana.	77.5-77.2%	Best 2L MLP +BP, 12 hid.	90.4%
5	PVM 1 rule	97.3%	**AIRS**	**94.9%**	SMART	76.8%	MLP +BP, 12 hid.	84.7%
6	**AIRS**	**96.7%**	C4.5	94.9%	GTO DT (5xCV)	76.8%	MLP +BP, 24 hid.	84.5%
7	FuNe-I	96.7%	RIAC	94.6%	ASI	76.6%	1-NN, Man.	84.2%
8	NEFCLASS	96.7%	SVM	93.2%	Fischer dis. ana.	76.5%	**AIRS**	**84.0%**
9	CART	96.0%	Non-lin. percept.	92.0%	MLP +BP	76.4%	MLP, +BP, 6 hid.	83.5%
10	FUNN	95.7%	FSM + rotation	92.8%	LVQ	75.8%	FSM - method?	83.6%
11			1-NN	92.1%	LFC	75.8%	1-NN Euc.	82.2%
12			DB-CART	91.3%	RBF	75.7%	DB-CART, 10xCV	81.8%
13			Lin. percept.	90.7%	NB	75.5-73.8%	CART, 10xCV	67.9%
14			OC1 DT	89.5%	kNN, k=22, Manh	75.5%		
15			CART	88.9%	MML	75.5%		
...					...			
22					**AIRS**	**74.1%**		
23					C4.5	73.0%		

over ten runs to allow for more direct comparisons with other experiments reported in the literature. During the experimentation, it was noted by the authors that varying system parameters, such as number of seed cells, varied performance on certain data sets; however, varying system resources (i.e., the numbers of resources an ARB could compete for) seemed to have little effect. A comparison was made between the performance of AIRS and other benchmark techniques, where AIRS seemed not to outperform specialist techniques, but did outperform more general purpose algorithms, such as C4.5. To save duplication, the reader is referred to [124] for a detailed account of classification accuracy comparisons.

Even though initial results from AIRS are promising, it can be said there are a number of potential areas for simplification and improvement. There is clearly a need to understand exactly why and how AIRS behaves the way it does. This can be achieved through a rigorous analysis of the algorithm, and through empirical examination of the behavior of the ARB pool and memory set over time.

Much of the focus in the investigation of the AIRS algorithm has been primarily on the classification performance. [51] performs some of this empirical exploration by applying AIRS to a variety of classification problems in which the number of class ranged from 3 to 12 and the number of features ranged from 4 to 279. In the course of this work, AIRS was found to have the best performance of any single classifier known to the authors on the publicly available credit.crx problem. Further empirical exploration of the AIRS algorithm is detailed in the description of two other suites of experiments: [50] discusses the effects of replacing the algorithm for evolving a candidate memory cell from the ARB pool and concludes

that most of the effectiveness of the classifier lies in the replacement strategies for the memory cell pool itself. [82] performs limited exploration of modifications to the resource allocation mechanism as well as a more thorough examination of the tie-breaking mechanism in the k-nn algorithm. In the course of this latter experimentation, it was found that AIRS outperforms the best reported accuracy for the E.coli data set found in the UCI repository [13].

The majority of AIS techniques use the metaphor of somatic hypermutation or affinity proportional mutation. The initial version of AIRS did not employ this metaphor but instead used a naïve random generation of mutations. The remainder of this chapter details investigations into the behavior of the algorithm and presents a modified version of AIRS, which is more efficient in terms of ARB production and employs affinity proportional mutation, and assess what, if any, difference these changes have made to the overall algorithm.

3.4 A More Efficient AIRS

The preceding sections of this chapter, with the exception of discussing AIRS in terms of our MAD learning framework, represented the state of AIRS as presented in [124]. For this thesis we follow the insights for extending this early work in order to produce a more efficient version of the algorithm. This section details observations that have been made through a thorough investigation into AIRS and how issues raised through these observations have been overcome.

3.4.1 Observations

The ARB Pool

A crude visualization was used to gain a better understanding of the development of the ARB pool.[4] In AIRS there are two independent pools of cells, the memory cell pool and the ARB pool. The original formulation of AIRS uses the ARB pool to evolve a candidate memory cell of the same class as the training antigen, which can potentially enter the memory cell pool. During this evolution, ARBs of a different class than the training antigen were also maintained in the ARB pool. The stimulation of an ARB was based both on affinity to the antigen and on class, where highly stimulated ARBs were those of the same class as the antigen which were "close" to the antigen, or were of a different class and "far" from the antigen. However, the visualization, which consisted of an animated GIF file which was the concatenation of a series of plots of the ARB pool at each iteration in the pool's evolution on simple 2-dimensional simulated data, revealed that, during the process of evolving a candidate memory cell, evolving ARBs that are of a different class than the training antigen was wasted effort. The point of the interaction of the ARB pool with the antigenic material is really only in evolving a good potential memory cell, and this potential memory cell must be of the same class as the training antigen. The visualization demonstrates that there is a process of convergence by ARBs of the same class to the training antigen. Naturally, based on the reward scheme, ARBs of a different class are moving

[4] http://www.cs.kent.ac.uk/people/rpg/abw5/ARB_hundred.html

further away from the training antigen. However, this process essentially must start over for the introduction of each new antigen, and, therefore, previously existing ARBs are fairly irrelevant. Since there are two separate cell pools, with the true memory of the system only being maintained in the Memory Cell pool, maintaining any type of memory in the ARB pool made no effective contribution. Eliminating the maintenance of multiple classes in the ARB pool while generating a candidate memory cell simplifies the algorithm, reduces the memory required during execution, and improves the overall runtime.

Mutation of Cells

Motivated by observing the success of other AIS work, as well as by some of the tendencies discussed in [124] and [127], attention was paid to the way in which mutation occurred within AIRS. In these two works, the authors notice that some of the evolved memory cells do not seem as high in quality as others. Additionally, it was observed that there seemed to be some redundancy in the memory cells that were produced. In [29] and other AIS work, mutation within an antibody or B-cell is based on its affinity, so that high affinity cells undergo mutation with a more restricted range than lower affinity cells. These other AIS works have used this method of somatic hypermutation to a good degree of success. It was thought that embedding some of this approach in AIRS might result in higher quality, less redundant, memory cells. This approach was therefore adopted within AIRS.

3.4.2 AIRS2: The Revisions

This subsection outlines the changes that have been made to the AIRS algorithm as part of this PhD thesis. We then follow with empirical results from the new formulation and a discussion of the implications of these results.

Memory Cell Evolution

In the original version of AIRS both ARBs "near" the antigen and of the same class as the antigen were rewarded and ARBS "far" from the antigen and of a different class than the antigen were rewarded. Also, ARBs were allowed to mutate their class values (mutate in this case means switching classes). In the newly revised version of AIRS, only ARBs of the same class are maintained in the ARB pool and mutation of the class value is no longer permitted. Figure 3.4.1 presents the changes to the algorithm presented in Figure A.2.3. The removal of resource allocation based on class is highlighted in **bold**.

Recall that the stimulation threshold was originally used as a stopping criterion for training the ARB pool on an antigen. In order to stop training on an antigen the average normalized stimulation level had to exceed the stimulation threshold for each class group of ARBs. That is, in a 2-class problem, for example, the average normalized stimulation level of all class 0 ARBs had to be above the stimulation threshold, and the average normalized stimulation level of all class 1 ARBs had to be above the stimulation threshold. It was possible, and frequently the case in fact, that the average normalized stimulation level for the ARBs of the

$minStim \leftarrow MAX$
$maxStim \leftarrow MIN$
foreach($ab \in AB$)
do
  $stim \leftarrow stimulation(ag, ab)$
  if ($stim < minStim$)
    $minStim \leftarrow stim$
  if ($stim > maxStim$)
    $maxStim \leftarrow stim$
  $ab.stim \leftarrow stim$
done
foreach($ab \in AB$)
do
  $ab.stim \leftarrow \frac{ab.stim - minStim}{maxStim - minStim}$
  $ab.resources \leftarrow ab.stim * clonal\_rate$
; no longer have multiple classes in the ARB pool
done
$resAlloc \leftarrow \sum_{j=1}^{|AB_{ag.c}|} ab_j.resources, ab_j \in AB_{ag.c}$
$NumResAllowed \leftarrow TotalNumResources$
; again no longer need to concern ourselves with
; multiple classes in the ARB pool
while($resAlloc > NumResAllowed$)
do
  $NumResRemove \leftarrow resAlloc - NumResAllowed$
  $ab_{remove} \leftarrow argmin_{ab \in AB_{ag.c}}(ab.stim)$
  if($ab_{remove}.resources \leq NumResRemove$)
    $AB_{ag.c} \leftarrow AB_{ag.c} - ab_{remove}$
    $resAlloc \leftarrow resAlloc - ab_{remove}.resources$
  else
    $ab_{remove}.resources \leftarrow ab_{remove}.resources - NumResRemove$
    $resAlloc \leftarrow resAlloc - NumResRemove$
done

Figure 3.4.1: Stimulation, Resource Allocation, and ARB Removal: Revised

same class as the training antigen reached the stimulation threshold before the average normalized stimulation level of ARBs in different classes from the antigen. What this did, in effect, was allow for the evolution of even higher stimulated ARBs of the same class while they were waiting for the other classes to reach the stimulation threshold. By taking out these extra cycles of evolution which were due to ARBs of different classes, it is possible that the ARBs will not have converged "as much" as in the previous formulation. This can be overcome by raising the stimulation threshold and thus requiring a greater level of convergence.

Somatic Hypermutation

To explore the role of mutation on the quality of the memory cells evolved, the mutation routine was modified so that the amount of mutation allowed to a given gene in a given cell is dictated by the cell's stimulation value. Specifically, the higher the normalized stimulation value, the smaller the range of mutation allowed. Essentially, the range of mutation for a given gene = 1.0 - the normalized stimulation value of the cell. Mutation is then controlled over this range with the original gene value being placed at the center of the range. This, in a sense, allows for tight exploration of the space around high quality cells, but allows lower quality cells more freedom to explore widely. In this way, both local refinement and diversification through exploration are achieved. This change is illustrated in figure 3.4.2, which is presented in a similar manner to figure A.2.2. The changes made are highlighted in **bold**.

```
mutate(x, b)
{
  range ← 1 − x.stim
  foreach(x.f_i in x.f)
  do
    change ← drandom()
    change_to ← drandom()
    bottom ← x.f_i / normalization_value − range / 2
    if (bottom < 0)
      bottom ← 0
    change_to ← (change_to ∗ range) + bottom
    if(change_to > 1)
      change_to ← 1
    if(change < mutation_rate)
      x.f_i ← change_to ∗ normalization_value
      b ← true
; no longer need to mutate class
  done
  return x
}
```

Figure 3.4.2: Mutation Routine: Revised

3.4.3 AIRS2: The Algorithm

The changes made to the AIRS algorithm are small, but end up having an interesting impact on both the simplicity of implementation and on the quality of results. Section 3.5 offers more discussion by way of comparison. For now, we discuss the changes to the original AIRS presented in section 3.2. These can be identified as follows:

- Only the Memory Cell pool is seeded during initialization rather than both the **MC** pool **(M)** and the ARB pool **(P)**. Since we are no longer concerned about maintaining memory or class diversity within **P** it is no

longer necessary to initialize **P** from the training data or from examples of multiple classes.

- During the clonal expansion from the matching memory cell used to populate **P**, the newly created ARBs are no longer allowed to mutate class. Again, maintaining class diversity in **P** is not necessary.

- Resources are only allocated to ARBs of the same class as the antigen and are allocated in proportion to the ARB's stimulation level in reaction to the antigen.

- During affinity maturation (mutation), a cell's stimulation level is taken into account. Each individual gene is only allowed to change over a limited range. This range is centered at the gene's pre-mutation value and has a width the size of the difference of 1.0 and the cell's stimulation value. In this way the mutated offspring of highly stimulated cells (those whose stimulation value is closer to 1.0) are only allowed to explore a very tight neighborhood around the original cell, while less stimulated cells are allowed a wider range of exploration. (It should be noted that during initialization all gene values are normalized so that the Euclidean distance between any two cells is always within 1.0. During this normalization, the values to transform a given gene to within the range of 0 and 1 are discovered, as well. This allows for this new mutation routine to take place in a normalized space where each gene is in the range of 0 and 1.)

- The training stopping criterion no longer takes into account the stimulation value of ARBs in all classes, but now only accounts for the stimulation value of the ARBs of the same class as the antigen.

3.4.4 Results and Discussion

To allow for comparison between the two versions of the algorithm, the same experiments that were performed on the original AIRS were performed on the new formulation of AIRS (AIRS2). Section 3.5 provides a more thorough comparative discussion, but for now, results of AIRS2 on the four previously discussed benchmark sets are presented in table 3.4.1.

Table 3.4.1: AIRS2 Classification Results on Benchmark Data

Iris	Ionosphere	Diabetes	Sonar
96.0%(1.9)	95.6%(1.7)	74.2%(4.4)	84.9%(9.1)

These results were obtained by following the same methodology as the original results reported in section 3.3, which is elaborated upon in [124] and [126]. Again, we note that these results are competitive with other classification techniques discussed in the literature, such as C4.5, CART, and Multi-Layer Perceptrons (as presented in table 3.3.1).

3.5 Comparative Analysis

This section briefly touches on some comparisons between the original version of AIRS presented in discussed in section 3.2 (AIRS1) and the revisions to this

algorithm presented in section 3.4. The focus of this discussion is on two of the more important features of the AIRS algorithms: classification accuracy and data reduction.

3.5.1 Classification Accuracy

The success of AIRS1 as a classifier (cf, [126]) makes it important to assess any potential changes to the algorithm in light of test set classification accuracy. To aid in this task, Table 3.5.1 presents the best average test set accuracies, along with the standard deviations (given in parentheses), achieved by both versions of AIRS on the four benchmark data sets. All experiments were repeated in the same way, using the same parameters as the original work.

Table 3.5.1: Comparative Average Test Set Accuracies

	AIRS1: Accuracy	AIRS2: Accuracy
Iris	96.7 (3.1)	96.0 (1.9)
Ionosphere	94.9 (0.8)	95.6 (1.7)
Diabetes	74.1 (4.4)	74.2 (4.4)
Sonar	84.0 (9.6)	84.9 (9.1)

It can be noted that the revisions to AIRS presented in section 3.4 do not require a sacrifice in the classification performance of the system. In fact, for three of the four data sets we see what appears to be a slight improvement in the accuracy. However, upon closer examination, we find that the differences in accuracy for these data sets are not statistically significant at the 99% ($\alpha = 0.01$) significance level based on a two-tailed t-test. Section B.1 gives a discussion on how this test is performed and how to interpret its results. Table 3.5.2 gives the

p-value results of this t-test comparing the mean accuracies for AIRS1 and AIRS2 on these four data sets which indicate the probability that the means are the same.

Table 3.5.2: t-test Results for Comparing AIRS1 and AIRS2 Accuracies

	n_1(AIRS1)	n_2(AIRS2)	p-value
Iris	15	15	0.46
Ionosphere	3	3	0.55
Diabetes	30	30	0.93
Sonar	130	130	0.44

In general, since $p \geq \alpha$, on these four data sets, we find that AIRS2 does not appear to introduce any adverse modifications to the behavior (or at least to the classification accuracy) of the classifier.

3.5.2 Data Reduction

From the previous subsection it can be seen that the changes introduced to AIRS offer no real difference in classification accuracy, so the question arises: why bother? Why introduce these changes to a perfectly reasonably performing classification algorithm? The answer lies in the data reduction capabilities of AIRS. In our study of the initial version of the algorithm, we found that aside from competitive accuracies another intriguing feature of the AIRS classification system is its ability to reduce the number of data points needed to characterize a given class of data from the original training data to the evolved set of memory cells [124, 127]. Given the volumes of data associated with many real-world data sets of interest, any technique that can reduce this volume while retaining the

salient features of the data set is useful. Additionally, it is this collection of memory cells that are the primary classifying agents in the evolved system. Since classification is currently performed in a k-nearest neighbor approach, for which classification time is dependent upon the number of data points, any reduction in the overall number of evolved memory cells is useful. Table 3.5.3 presents the average size of the evolved set of memory cells and the amount of data reduction this represents in terms of population size and percentage reduction, along with standard deviations, for each version of the algorithm on the four benchmark data sets. The original training set size is also presented for comparison. There are two points of interest: 1) Both versions of the algorithm exhibit data reduction, and 2) AIRS2 tends to exhibit greater data reduction than AIRS1. Naturally, this data reduction is domain dependent, and there is no guarantee that this trend would continue to all data sets. However, this observation about data reduction is significant for our current discussion. As mentioned in subsection 3.4.3, one of the goals of the revision of the AIRS algorithm is to see if employing somatic hypermutation through a method more in keeping with other research in the AIS field would increase the efficiency of the algorithm. The current measure of efficiency under concern is the amount of data needed to represent the original training set to achieve accurate classifications. We can see from Table 3.5.3 that, for the majority of the data sets, AIRS2 is able to achieve accuracy comparable with AIRS, with greater efficiency. This observation is made by comparing the size of the training data set with the number of developed memory cells in the final classifier. Table 3.5.4 shows that the differences in average memory cell size

Table 3.5.3: Comparison of the Average Size of the Evolved Memory Cell Pool

Training Set	Size	AIRS1: Memory Cells	AIRS2: Memory Cells
Iris	120	42.1/65% (3.0)	30.9/74% (4.1)
Ionosphere	200	140.7/30% (8.1)	96.3/52% (5.5)
Diabetes	691	470.4/32% (9.1)	273.4/60% (20.0)
Sonar	192	144.6/25% (3.7)	177.7/7% (4.5)

is significant at the 99% significance level ($\alpha = 0.01$) for all four data sets using a t-test as described in section B.1. For half of the data sets, Ionosphere and Diabetes, the degree of data reduction is greatly increased (from 30% to 52% for Ionosphere data and from 32% to 60% for the Diabetes data set). Interestingly, for the most difficult classification task, the Sonar data set, the degree of data reduction is not increased, but rather decreased. So while it may be reasonable to claim that in general it appears that the revisions to AIRS provide greater data reduction, and hence greater efficiency, without sacrificing accuracy, this cannot be blindly accepted as a characteristic of AIRS2. As with all learning algorithms, the domain to be learned has a greater influence on the performance of the algorithm than any other factor.

Table 3.5.4: t-test Results for Comparing AIRS1 and AIRS2 Memory Pool Sizes

	n_1(AIRS1)	n_2(AIRS2)	p-value
Iris	15	15	$p < 0.01$
Ionosphere	3	3	$p < 0.01$
Diabetes	30	30	$p < 0.01$
Sonar	130	130	$p < 0.01$

3.5.3 Asymptotic Analysis

We end this section with a discussion of the asymptotic behavior of AIRS. We start by analyzing the behavior of the initial version of AIRS and conclude how the changes presented in section 3.4 (AIRS2) impact the requirements of the algorithm.

For this analysis, we employ the following notation, let:

- N be the number of training items;
- L be the number of features in each training item;
- M_i be the number of memory cells present for training antigen i;
- A_j be the number of ARBs present during generation j;
- G_i be the number of Generations of ARBs that must be produced to reach the stimulation threshold when training on antigen i;
- R be the number of resources allowed in the system;
- C be the clonal rate;
- and let T be the number of data items to be classified.

The initialization stage requires the calculation of the Affinity Threshold (AT). As indicated in equation, A.2.1 this is achieved through pairwise calculation of the affinity between each training data item.[5] This will require $\frac{N*(N-1)}{2}$ steps

[5]Some experiments have been performed with choosing only a random subset of the training set for use in calculating the AT. These experiments seem to indicate that limiting the calculation of the AT to only a fraction of the training set has no adverse effect on the performance of the algorithm.

to calculate or $O(N^2)$. Assuming that the affinity between any two items can be calculated in linear time in terms of the number of features L, $O(L)$, then calculation of the AT is $O(LN^2)$.

For the training stage, the worst case complexity will occur when all memory cells and training antigens are of the same class. This requires a search through the entire memory cell pool for each antigen in order to find mc_{match}. While, admittedly, this may make no sense from a classification standpoint, it will allow us to establish a worst case behavior. So, for each of the N training antigens, we perform the following steps:

1. The location of mc_{match} requires calculating the affinity between the antigen and each of the memory cells. This step is, then, $O(LM_i)$.

2. The production of mutated offspring by mc_{match} is in proportion to the number of features L, since each feature in each clone is given an opportunity to mutate. In actuality, the constants of the clonal rate and hypermutation rate dictate how many mutated offspring mc_{match} are allowed to produce. However, these user-defined constants are quite often less than the number of features in the data vectors. Therefore, asymptotically, this step costs $O(L)$.

3. G_i times do:

   (a) Calculate stimulation value for each ARB in reaction to the antigen: $O(LA_j)$.

(b) Allocate resources to each ARB: $O(A_j)$.

(c) Sort ARBs by stimulation value: $O(A_j \log_2 A_j)$.

(d) Find and remove dead ARBs: $O(A_j)$.

(e) Calculate the average stimulation value of the still living ARBs. Since the maximum number of ARBs still alive at this point is dictated by the number of resources, R: $O(R)$.

(f) Produce mutated offspring of the ARBs: $O(LR)$.

4. The choosing of the most stimulated ARB to be $mc_{candidate}$ requires constant time since the ARBs have already been sorted based on stimulation value. This step, then, is $O(1)$.

5. Finally, determining if $mc_{candidate}$ should be added to the memory cell pool and if it should replace mc_{match} requires calculating the affinity between the antigen and $mc_{candidate}$ and between mc_{match} and $mc_{candidate}$. Therefore, this step takes $O(L)$.

Given this, the asymptotic on the training routine would be:

$$Tr(N) = O(\sum_{i=1}^{N}(LM_i + L + (\sum_{j=1}^{G_i}(LA_j + A_j \log_2 A_j + LR)))) \tag{3.5.1}$$

There are three terms in this analysis of the training routine that need further clarification: M_i, A_j, and G_i. If we are following our worst case assumptions, then we can assume that the the memory cell pool was initially seeded with all N training data items. This would make $M_i = N$ in the worst case.

With the exception of the first iteration, the number of ARBs in the ARB pool at each iteration, A_j, is dependent on the number of resources available in the system, R. The number of resource each ARB is allocated is based in the ARBs stimulation level and a user-defined parameter: clonal rate, C. If an ARB is maximally stimulated (stimulation level of 1), then it is allocated C resources. If it is minimally stimulated (stimulation level of 0), then it receives 0 resources and is removed from the system. For our bounding cases, however, let us first consider a set of ARBs that are all maximally stimulated. In this case, the number of ARBs in the ARB pool before cloning would be $\frac{R}{C}$. This is the minimum number of ARBs in the pool at any time. Next, let us consider a set of ARBs that are each allocated only one resource. In this case, the number of ARBs in the ARB pool would be R. The number of ARBs before cloning, then, is bounded as follows:

$$\frac{R}{C} \leq \text{Number of ARBs} \leq R \tag{3.5.2}$$

These ARBs are then allowed to clone. The number of clones an ARB is allowed to produce is based on its stimulation level. Each ARB can create up to C mutated offspring. (In reality, only a random subset of the ARBs are allowed the opportunity to produce offspring, but for now we will assume that all ARBs still alive are given this chance.) Returning to our bounding cases, recall that we have $\frac{R}{C}$ ARBs before cloning if all of these ARBs were maximally stimulated. If they were all maximally stimulated and they all produced C mutated clones, then we

would have $A_j = \frac{R*C}{C} + \frac{R}{C}$ or $A_j = R + \frac{R}{C}$.[6] That is, we have the original $\frac{R}{C}$ ARB parents that survived and the new R mutated clones that they created. In our other case, when every ARB was only allocated one resource based on their low stimulation levels, we can assume that each ARB is allowed to create at most one clone. In this case, $A_j = 2 * R$. However, since not every ARB is allowed to create mutated clones and even those ARBs that are allowed to produce clones may not produce their full allotment, these will only produce loose bounds. In fact, for our lower bound, we assume that none of these highly stimulated ARB produce any clones. Still, A_j can be bounded as follows:

$$\frac{R}{C} < A_j < 2 * R \tag{3.5.3}$$

Therefore, asymptotically $A_j = O(R)$.

This leaves G_i, the number of generations (or iterations) of ARBs that need be produced before the stopping criterion is met, as the final term in equation 3.5.1 that needs further clarification. The stopping criterion is detailed through equation A.2.4 in section A.2.3. Unfortunately, there is no straight-forward way to specify for any problem how long it will take for this stopping criterion to be met. It is dependent on several factors:

[6] In actuality, this scenario could never occur since if all of the ARBs were maximally stimulated our stopping criterion would be met and there would not be another pass through the ARB pool with this particular antigen.

- The mutation rate dictates how much each offspring varies from its parent cell. In AIRS1, each cell was allowed to mutate freely whereas in AIRS2 this mutation range was fixed based on the parent cell's stimulation value.

- The clonal rate C dictates how many mutated offspring each cell is given the opportunity to produce.

- Obviously, the stimulation level of a given cell is the most important factor for which cell gets selected to create offspring. This is the heart of the clonal selection and the only reason that the ARB pool will drift toward the stopping criterion at all. The most stimulated ARB is given the opportunity to create the largest number of mutated offspring.

- The number of resources, R, dictates how many ARBs survive from generation to generation. There should exist a trade-off between the value of R and the quality of solution evolved by AIRS. If R is too low, the stopping criterion may be swiftly met, but $mc_{candidate}$ might be of such poor quality that it would never be added into the memory cell pool. If R is too high, then the ARB pool may continue to be too diverse to reach the stopping criterion.

From this discussion, it is clear that the number of resources plays a large role in the overall run-time performance of the algorithm. Interestingly, experiments have shown that the accuracy of AIRS is not incredibly sensitive to small perturbations of this value.

So, we can now simplify our training time analysis:

$$Tr(N) = O(\sum_{i=1}^{N}(LM_i + L + (\sum_{j=1}^{G_i}(LA_j + A_j\log_2 A_j + LR))))$$

$$Tr(N) = O(LN^2 + \sum_{i=1}^{N}(G_i(LR + R\log_2 R))) \quad (3.5.4)$$

It should be noted, however, that in practice M_i would almost never actually be N. This is due to the fact that for AIRS to be effective it would not be seeded with all N training times.[7] Also, AIRS is not typically run on a problem that contains only one class which would be the requirement for $M_i = N$. Nevertheless, equation 3.5.4 does give a definite upper bound on the runtime behavior of the training routine.

Finally, in the worst case, the memory cell pool contains on the order of N memory cells at the end of training. Since classification is performed using a k-NN approach with the same affinity calculation used throughout, the classification of T data items would require $O(LTN)$.

As mentioned in section 3.4, the reformulation of AIRS was chiefly motivated by some basic observations about the workings of the system. One observation was that the original version of AIRS maintained representation of too many cells for its required task. This led to the elimination of maintaining multiple classes of cells in the ARB pool or of retaining cells in the ARB pool at all. This has the simplifying effect of reducing the memory necessary to run the system

[7] AIRS is simply k-NN in this case.

successfully. A second observation concerning the quality of the evolved memory cells led to the investigation of the mutation mechanisms employed in the original algorithm. By adopting an approach to mutation proven to be successful in other AIS, it has been possible to increase the quality of the evolved memory cells that is evidenced by the increased data reduction without a decrease in classification accuracy. Both of these overarching changes (ARB pool representation and the mutation mechanisms used) have exhibited a simplifying effect on the classification system as a whole. While our analysis in this section has focused on the runtime complexity rather than the memory requirements of AIRS these, two factors go hand-in-hand. The reduction in memory requirements is seen primarily in the reduced number of memory cells that must be searched to find MC_{match}. That is, the term M_i in equation 3.5.1 is decreased. Additionally, the difference in quality of the cells evolved through this process should also have an impact on the number of generations required before reaching convergence. That is, the G_i term in our analysis equations. In general, the changes made in the development of AIRS2 leads to an overall simplification of the algorithm.

3.6 Summary

This chapter has detailed an immune-inspired learning algorithm called AIRS. From a learning standpoint, AIRS is one of the first supervised learning algorithms based on immunological ideas. In this chapter we explained the immunological metaphors underpinning the AIRS algorithm. We examined the performance of

AIRS on some standard machine learning data sets, which revealed the relative success of this approach. Through basic algorithm modeling and a return to the biological inspiration of the system, two primary areas of simplification for this algorithm were identified, and these basic changes were made. Statistical evidence suggests that these changes did not affect the classification accuracy of the algorithm and did improve the data reduction capabilities (and thus efficiency) of the algorithm. We concluded with an asymptotic analysis of the run-time behavior of the algorithm.

Since its inception, the AIRS algorithm has gained a certain amount of use in the AIS community. Marwah has investigated some of the fundamental assumptions embedded in the classification routine of the algorithm [82]. Goodman examined the classification ability on multi-class problems [51] and searched for the source of classification power in AIRS [50, 49]. Greensmith explored the use of AIRS for semantic web classification [54]. Finally, Hamaker has introduced the use of different affinity functions and datatypes into AIRS [56].

Having illustrated how a return to the biology can improve this immune-inspired system, we move back to standard computing techniques for our next enhancement. We now examine how the use of parallel computing techniques can be incorporated in immune-inspired learning algorithms for decreased runtimes. As with the initial formulation of AIRS, this step is motivated foremost by the lack of any real investigation into the use of more processing power with immune-inspired algorithms. We begin this portion of our study by first providing an overview of parallel computing and parallel performance

metrics. We then examine parallel genetic algorithms for common methods of parallelizing population based algorithms. To establish the efficacy of our proposed parallelization methods, we employ these techniques on a very basic immune-inspired algorithm, the clonal selection algorithm CLONALG. This then leads to our examination of a parallel version of the AIRS learning algorithms.

Chapter 4

Parallelizing AIS Learning Algorithms

Among the oft-cited reasons for exploring verterbrate immune systems as a source of inspiration for computational problem solving include the observations that the immune system is inherently parallel and distributed with many diverse components working simultaneously and in cooperation to provide all of the services that the immune system provides [28, 25]. Within the AIS community, there has been some exploration of the distributed nature of the immune system as evidenced in algorithms for network intrusion detection (e.g., [61, 72]) as well as some ideas for distributed robot control (e.g., [78, 77]), to name a number of examples. However, very little has been done in the realm of parallel AIS—that is, applying methods to parallelize existing AIS algorithms in the hopes of efficiency (or other) gains. This focus on computational efficiency in system parallelization is slightly different than the goals found of distributed computing.

In distributed computing, the impetus tends to be an exploration of the use of computational resources for increased diversity of reaction or on problem solving in a highly decentralized manner where each computational resource requires independent decision making facilities with little to no input from a centralized mechanism. In a parallel system, however, problem solving is much more tightly coupled and the use of additional computational resources tends to be motivated by decreasing overall system runtime or increasing efficient system utilization The exploitation of parallelism inherent in many algorithms has provided definite gains in efficiency and lent insight into the limitations of the algorithms [18, 22, 47, 20]. While just parallelizing AIS algorithms is, admittedly, venturing fairly far afield from the initial inspiration found in the immune system, the computational gains through this exercise could well be worth the (possible) side-track. Additionally, this exploration may provide some insight into other relevant areas of AIS, such as the development of new models or as ways to incorporate diversity or even understanding the need for such into our current models.

This chapter begins by introducing basic methods and metrics of parallelization. This provides a high-level overview of the terminology and concepts frequently encountered when discussing parallel algorithms. We follow this with a discussion of parallel Genetic Algorithms. Since GAs are population-based evolutionary algorithms, there may be some similarities between the methods for parallelizing these algorithms and our population-based immune algorithms. We conclude this chapter with an outline of how to apply these ideas to immune learning algorithms.

4.1 Introduction to Parallelism and Parallel Performance Metrics

Our fundamental goal when employing parallelism and parallelizing a given algorithm is a speed-up in overall time. That is, we are most concerned with the computational performance of the parallel algorithm. This is in contrast to the goals of distributed computing which tend to focus more on the use and provision of distributed resources rather than performance. While there is a definite relationship between these two methods of utilizing multiple computational resources (as opposed to a single, serial resource), the goals of the two approaches are slightly different. For the next few chapters we focus primarily on parallel computing. We do not mean to imply that we are oblivious to the other qualitative changes that may occur when we parallelize our given immune algorithms; however, our chief initial concern will be the speed we gain through our parallelism. We return to the concerns unique to distributed computing in chapter 7.

While we do not detail here the architectural concerns of parallel computers, we do want to point out that our studies have focused solely on the use of a multiple-instruction multiple-data (MIMD) architecture, to use Flynn's classic taxonomy [37, 41].[1] In even more practical terms, we have utilized a cluster of

[1]It should be noted that the immune system has the potential to teach us a lot about possible parallel architectures. Exploration of the distributed control mechanisms observed in biological immune systems to gain such insights is needed. However, we currently set aside these architectural issues while briefly discussing them in chapter 7

interconnected serial computers to execute our parallel codes using a message passing paradigm. What this implies for our further analysis is that we must account for communication time among the processes when we discuss our performance metrics.

Regardless of architectural concerns, there are several fundamental performance metrics that we can discuss in terms of parallel codes.[2] Since, as we mentioned previously, what we are ultimately concerned with is the speed we gain through parallelism, the most basic measure of this is speedup (i.e., the ratio of the serial execution time (t_1)to the parallel execution time(t_p)):

$$S = \frac{t_1}{t_p} \tag{4.1.1}$$

Our goal in parallelizing an algorithm could be seen as maximizing this ratio. If our serial algorithm can do w work in w seconds (i.e., $t_1 = w$) and if we assume that we can arbitrarily assign the instructions in our serial algorithm to each of the p processors and that we can do this evenly across our processors, then we end up with an upper bound on speedup S_u:

$$S_u = \frac{w}{\frac{w}{p}} = p \tag{4.1.2}$$

This "linear" speedup gives us an ideal on our goal of trying to maximize S. While this gives us a theoretical ideal when only accounting for processor time

[2]The discussion of these basic metrics has been paraphrased, in large part, from [81].

and w, it is possible to observe "superlinear" speedup results in experimental studies. One possible source of this superlinear speedup derives from equation 4.1.2 ignoring architectural concerns such as the structure of the memory system and time required to access data in this system. In a multiprocessor environment, and particularly in a cluster of computational resources, each processor has its own memory system and local cache. Superlinear speedup can result from the parallel version of the code needing less time to access memory locations than the serial version (e.g., more of the data needed is in cache rather than on disk). While this access time is implicit in w in equation 4.1.2 for the serial version of the algorithm, it is not fully captured in the parallel version if access to the same data is not constant when compared to the serial version.

While speedup provides us a simple metric of overall performance, we are often concerned, as well, with how well we utilize our parallel resources. Or, we are concerned with what losses in efficiency our parallelization of our code has introduced. A basic way to measure this is through parallel efficiency (E_p):

$$E_p = \frac{S}{S_u} = \frac{t_1}{p * t_p} \tag{4.1.3}$$

The ideal value for E_p, then, would be 1 which is achieved when the number of processes is equal to 1, $p = 1$. However, if $p = 1$, then we will not have achieved any speedup in computational time. Frequently, in the design of parallel algorithms there is a trade-off between performance (as measured in S) and resource usage (as measure in E_p).

Apart from the performance gains speedup indicates and the utilization measures that parallel efficiency provides, we would also like a way of determining how scalable a given parallel system is. By scalable we mean that the efficiency of the system can be fixed to some constant for increasing the number of processors if we also increase the amount of work to be done [76]. This becomes a necessary discussion since, according to Amdahl's law, if we have a fixed problem size then the speedup of the algorithm will not continue to increase with an increase in the number of processors [5]. Determining the scalability of a system can allow us to predict how many processors we can use before we begin to see performance degradation as well as the size of the problem that our system can handle. Kumar and colleagues have defined the isoefficiency function as a means of measuring this concept of scalability [52, 53]. The basic idea behind this tool is to determine the impact of the parallel overhead on the overall performance of the system and to determine at what rate the size of the problem must grow in order to take advantage of an increase in processor power. Central to this concept is a slightly different examination of speedup and efficiency. Rather than being solely defined as in equation 4.1.1, Kumar et al. first observed that the parallel time, t_p can be defined as [52]:

$$t_p = \frac{t_1 + t_o}{p} \tag{4.1.4}$$

where t_o is the parallel overhead time (i.e., time for communication among the processors, communication setup, etc.). Speedup can then be defined as:

$$S = \frac{pt_1}{t_1 + t_o} \tag{4.1.5}$$

and efficiency as:

$$E = \frac{t_1}{t_1 + t_o} = \frac{1}{1 + \frac{t_o}{t_1}} \tag{4.1.6}$$

The benefit of this slight reformulation is that it then allows us to discuss the performance of a parallel system in terms of its parallel overhead, t_o, which in turn allows us to determine bounds on this overhead that we can use to examine the limits of scalability within a system.

As we examine the specific implementations of parallel immune learning algorithms we return to these metrics and concepts. In particular, we measure the speedup and efficiency of our various newly formulated algorithms and try to determine what they tell us about the overall scalability of these systems.

4.2 Parallel Genetic Algorithms

Since we can argue that AIS can (in some respects) fit into the category of evolutionary algorithms, one of the natural starting places for the investigation of parallel AIS is the ways other evolutionary approaches have been parallelized. There have been numerous studies of parallel genetic algorithms, but the best

place to start is in two works by Erik Cantú-Paz [17, 18]. In these works the author examines the various forms of parallel GAs in great detail and offers the following classification of parallel GAs:

- Global or Master-Slave—This form of parallel GA exhibits the same characteristic as a serial GA. What is typically parallelized is the calculation of fitness values. A master process farms out given chromosomes to different processes where the fitness is calculated and then returned to the master process. The master process continues to perform the other genetic operators such as selection, crossover, and mutation;

- Coarse-grained—This form of parallelism employs the use of a few, relatively large demes (or isolated population groups) and migration. Each deme is allowed to evolve separately and then at certain points migration of select individuals occur from one deme to another;

- Fine-grained—This is similar to coarse-grained, except the population size in each deme is much smaller and the migration occurs more frequently;

- Hybrid—This employs various combinations of the three typically in a hierarchical manner with a coarse-grained approach at the top of the hierarchy.

The most widely explored option has been the coarse-grained approach [110, 111, 79]. The key factors in this approach are the migration policies—this includes the migration interval (how frequent migration occurs), the migration rate (the

number of individuals that migrate), and the manner in which individuals are chosen to migrate—and the topology of the processes (which processes are allowed to communicate with which others or the patterns of migration across the various processes). For many of the coarse-grained approaches these factors are static decisions made prior to running; however, in [95] we see the use of more adaptive policies based on various criteria. In general, however, migration is utilized to keep each isolated population from converging too quickly to a local optimum and in order to increase the diversity in each population.

In addition to outlining some of the parallelization strategies of GAs, Cantú-Paz also provides us with methods for analyzing the parallel behavior of these different techniques [18]. These methods do not go much beyond the application of the basic metrics presented in section 4.1, but they do provide a nice underlying model for analyzing the behavior of parallel population based algorithms. As we mention in section 4.3, however, the analysis of parallel GAs are made somewhat more tractable by the procedures common across all genetic algorithms (i.e., crossover, mutation, fitness evaluation, selection of fittest individuals for reproduction, etc.) than the large variety of procedures used in AIS. Additionally, much of the theory of GAs has been predicated on the use of binary chromosomes (or feature vectors) for the problem space representation. This assumption of binary representation has quite often not held true in the field of AIS. Nevertheless, [18] provides a useful study of parallel GAs that can provide insight in our parallelization of AIS algorithms.

4.3 Parallel Immune Learning

Turning our attention, now, to parallel AIS in general, we would like to establish a framework for discussing methods of parallelizing AIS. One of the difficulties in discussing AIS in general is that, other than modeling, or using, some aspect gleaned from the immune system, there are few, if any, commonalities across all AIS. GAs have the benefit of, regardless of application area, having some fairly clearly defined stages: fitness calculation, chromosome selection, mutation, and crossover. With these common stages for almost all GAs, it becomes easy to abstract which stages are exploitable for parallelization. With AIS, however, there are numerous algorithms that attempt to utilize different immune mechanisms to approach a given problem. So, it is necessary to limit the scope of this discussion to only certain immune algorithms that utilize certain principles from the immune system.

Naturally, as throughout this thesis, we only concern ourselves with immune-inspired learning algorithms. However, as evidenced in section 2.3, just the restriction of learning algorithms is probably not sufficient given the wealth of metaphors employed. We can start with the clonal selection models (to borrow a term from [28]), which do seem to have some stages in common. These stages could include antigenic presentation, affinity evaluation, clonal selection, and affinity maturation (to name a few), each of which might be broken down into further stages, yet. One area that needs to be addressed is which of these stages would most readily lend themselves to parallelization. For GAs, we saw that

for the global variety of parallelization it was the calculation of fitness that was parallelized. However, for the other schemes, entire GAs occupied each processor with some degree of interaction across the processes typically at the crossover or selection stages. This latter strategy could be adopted to several AIS. We could easily allow some of the dynamics of a given AIS to take place as isolated processes with a degree of communication among the evolving systems such as some type of cellular exchange.

To begin with, we limit our field of study even further to population-based, clonal selection AIS models. By restricting ourselves to population-based algorithms (as opposed to network-based ones), we have models that more closely resemble GAs and have less, in theory, inherent interaction than the network models. Unlike with GAs, current AIS tend to use fairly cheap "fitness" or affinity functions. This implies that the master-slave model mentioned in the previous section (section 4.2), in which it is the fitness evaluation that is parallelized, would not necessarily give us much benefit. However, the coarse-grained model seems an ideal place to begin. If we identify portions of an AIS algorithm that require little or no global interaction, we can isolate those parts to a given processor. The various populations can develop through the immune mechanisms at each processing node, and then a global solution can potentially be reconstructed if such is needed.

Aside from exploiting some of the parallelism inherent in our immune-based algorithms and thus giving us, hopefully, computational gains, this method also has the benefit of appearing somewhat more biologically plausible. The immune

system, as previously mentioned, is a distributed system with much occurring in isolated locations rather than a central immune response location. This "coarse-grained" approach allows us to partially explore this idea. Still, while this may be a useful side-effect of our adopting the coarse-grained approach, what we are primarily interested in initially is computational gains to be made through parallelization.

One of the issues, however, with this approach is that most immune-inspired learning algorithms have some eventual global component that is used to assess test data or previously unseen data. If we distribute a portion of the algorithm to isolated processes, we eventually have to tackle the issue of recapturing this global portion used for testing. For this reason, when we examine our parallelized immune algorithms we want to assess the impact this parallelization has on the quality of the learned solution as well. That is, we need to determine if the learned memory and decision-making structure can function as well as the serial versions. Additionally, if we find it functions better than in the serial versions, then this becomes important to us, as well.

4.4 Summary

This chapter has provided a transition from our discussion of immune learning in general and our in-depth look at one particular serial learning algorithm to the idea of parallel immune learning. That is, we have moved from our biological inspiration to the computational inspiration in the development of our

learning systems. We began by introducing some of the terminology relevant to discussing parallel algorithms and in particular parallel performance. This section emphasized the fact that what we are most concerned with in our parallelization is performance gain. We then moved on to discuss parallel GAs, and offered a summary of the taxonomy of such algorithms presented in [18]. This led to our discussion of potential means of parallelizing immune-inspired learning algorithms. We quickly encountered the problem that there is no general AIS to discuss (unlike the existence of a generic GA). Therefore, we resolved to only focus on population-based immune-learning algorithms.

In the following two chapters examine the effects of applying a "coarse-grained" model of parallelizing two immune-inspired learning algorithms. We begin with a simple immune-inspired algorithm (CLONALG) to demonstrate the applicability of parallel-computing techniques to these types of learning systems. We then return to our primary exemplar system, AIRS, and examine the modifications needed to exploit greater processing power in this bio-inspired algorithm. Our initial assessment is in terms of speedup and efficiency and is examined empirically through the use of bench-mark datasets. Then, for each of our parallel algorithms, we provide experiments that allow us to assess the scalability of the systems. We conclude our studies of using multiple processors for immune-learning algorithms by offering a very basic distributed approach. Unlike our parallel models whose chief concern is parallel performance, the distributed model points towards areas for the incorporation of a more biologically plausible use of different processing sites.

Chapter 5

The Clonal Selection Algorithm, CLONALG

This chapter explores the application of parallel processing techniques to an immune-inspired algorithm.[1] We offer this chapter mostly as a "proof-of-concept" type of approach to the use of parallel computing in our learning algorithms. To this end, this chapter examines a second, simpler, immune-inspired algorithm in some detail, which offers us an ideal test bed for our initial explorations of immune learning in parallel. In [32], the authors introduce an artificial immune system algorithm inspired primarily by the clonal selection theory first popularized by Burnet [16]. This algorithm, CLONALG, utilizes the immunological concepts of memory cells, free antibodies, clonal selection, and affinity maturation to provide solutions for pattern recognition and function optimization tasks. While we

[1] A much shorter version of this discussion was published as [125].

provide an overview of CLONALG, for a complete specification of this algorithm, please see [32].

Much like AIRS (chapter 3), CLONALG's primary goal is in the development of memory cells which provide some representation of the data set. Similar to many evolutionary algorithms, CLONALG proceeds through a number of generations in the development of this representation. Additionally, there are several user-tunable parameters which affect the rate of convergence in this evolutionary process. While we are interested in CLONALG for its application of immune-system principles to machine learning, what really intrigues us is the almost embarrassingly parallel nature of the algorithm. Since the algorithm is one of the more simplistic immune-inspired models, this is an excellent starting place for our exploration of parallel immune-inspired learning algorithms.

We begin this chapter with an overview of the serial version of the CLONALG algorithm as presented in [32] along with an analysis of the runtime characteristics of the algorithm. We conclude section 5.1 with a discussion of the changes that need to be made for parallelizing the algorithm. In order to assess the benefits of our parallel version of the algorithm, we repeat the experiments in [32] using our new algorithm. These results are discussed and presented in section 5.2. We follow these verification experiments with a series of simulations in which we vary the number of features and the number of instances to be learned. These experiments, presented in section 5.3, help us assess the scalability of the parallel version of the algorithm. We conclude this chapter with a reflection on the lessons learned from this initial parallel immune learning algorithm.

5.1 Description of CLONALG and its Parallelization

We begin this section with an overview of the serial version of CLONALG. This is followed by an analysis of the algorithm and a scheme for parallelizing this immune-inspired algorithm.

5.1.1 Serial CLONALG

CLONALG, as presented in [32], has two incarnations: one for pattern recognition tasks and one for multi-modal function optimization. Since the developers of this algorithm have subsequently focused primarily on the function optimization version, we have decided to explore the pattern recognition version more thoroughly. This version of the algorithm proceeds in a similar fashion to many evolutionary algorithms; however, the inspiration for the algorithm is clearly from the realm of immune systems. The input (or antigens) to the algorithm consists of a collection of $\mathbf{S}$ patterns to be recognized. The algorithm begins by generating a random population ($\mathbf{P}$) of N cells (or antibodies), where $|\mathbf{P}| \geqq |\mathbf{S}|$. For the current application, these cells simply consist of a binary feature string that represents a given binary pattern. A pattern from $\mathbf{S}$ is presented to each cell in $\mathbf{P}$, and the affinity of each cell to the pattern is calculated. Affinity is based on hamming distance as detailed in equation 5.1.1, where l is the length of the binary string feature vector and $x.i$ and $y.i$ refer to the ith feature of this vector in each of the

two cells being compared:

$$\text{affinity}(x, y) = \sum_{i=1}^{l} \begin{cases} 1 & : \quad \text{if } x.i \neq y.i \\ 0 & : \quad \text{otherwise} \end{cases} \tag{5.1.1}$$

Based on this equation, we see that affinity is measured simply as the number of different bits in the antibody when compared to the input pattern, or antigen. Therefore, a lower affinity number implies a closer match to the input pattern.

Once the affinity is calculated for each antibody in $\mathbf{P}$ as compared to a given input pattern, the best $n1$ antibodies are selected to undergo somatic hypermutation as a mechanism for providing affinity maturation. By "best" antibodies, we mean those which are closest in matching to the input pattern (i.e., those with the lowest affinity measure based on equation 5.1.1). The authors of [32] do not specify exactly what mechanism should be used for the mutation routine. However, they do specify that the mutation should be dependent on the affinity value of the antibody in two ways: 1) the number of mutated offspring a given cell is allowed to produce is dependent on the affinity of the cell with the input pattern; the more closely a cell matches the input pattern the more offspring it is allowed to produce and 2) the closer the match of the antibody to the given pattern the less mutation of the features should occur. So, we have a balance of exploration and exploitation that we see so often in machine learning algorithms. CLONALG encourages the exploitation of the good solutions it has

found by allowing high affinity cells to produce more offspring. It encourages exploration by increasing the mutation rate for low-affinity cells.

After the mutated offspring cells are generated, the input pattern is then presented to each of the newly created offspring, and their affinity levels are calculated. The single best antibody is then compared to the memory cell for the given input pattern. If this antibody's affinity is better than that of the memory cell, the newly generated antibody replaces the established memory cell. Finally, $n2$ random antibodies are created and replace existing cells in the population $\mathbf{P}$ of antibodies. This process continues until all the input patterns have been presented to the population.

One cycle through the input patterns is considered one generation. The algorithm can continue for a fixed number of generations or until a certain convergence criterion is reached. For the current experiments, the algorithm is only halted when absolute convergence occurs. By absolute convergence, we mean that the memory cell set which has evolved exactly matches the input pattern set. While in real-world learning situations this convergence criterion would not be as useful as some fuzzier threshold which would allow for greater generalization by the system, for our current experiments that examine the behavior of the algorithm as various parameters are adjusted we felt this was a sufficient stopping criterion.

The algorithm is presented below, in terms of the immune processes employed.

1. *Initialization*: Create a random population of N antibodies $\mathbf{P}$ and memory cells $\mathbf{M}$.

2. *Antigenic Presentation*: for each antigenic pattern in $\mathbf{S}$ do:

   (a) *Clonal Expansion*: Determine the affinity of each antibody in $\mathbf{P}$ to the given antigenic pattern. Select the best $n1$ antibodies to produce clones in proportion to its affinity.

   (b) *Affinity Maturation*: Mutate each clone based on its affinity.

   (c) *Clonal Selection*: Choose the best clone as a candidate memory cell. Replace the memory cell in $\mathbf{M}$ if the clone has a better affinity.

   (d) *Metadynamics*: Replace $n2$ antibodies in $\mathbf{P}$ with randomly generated cells. The antibodies chosen to be replaced are those with the worst match (i.e., highest value according to equation 5.1.1) in population $\mathbf{P}$.

3. *Cycle*: Repeat step 2 for a fixed number of generations or until the memory cells in $\mathbf{M}$ have converged to within some threshold of the input patterns in $\mathbf{S}$

Figure 5.1.1 provides the pseudocode for the version of CLONALG studied.

For clarity, we repeat the application of our MAD framework to CLONALG from section 2.3.2:

- Memory: The memory system consists of a group of evolved memory cells which represent patterns to be recognized;

```
generate a random population of N strings called P

foreach(generation)
do
  foreach(s ∈ S)
; S is the set of patterns to be recognized

  do
    foreach(p ∈ P)
    do
      calculate_affinity(p,s)
    done
    P1 ← P
; P1 is used for sortingP

    sort P1 based on affinity
    C ← "best"n1cells inP1
    clone_and_mutate(C)
; the best n1 are allowed to produce mutated clones

    foreach(c ∈ C)
    do
      calculate_affinity(c,s)
    done
    sort C based on affinity
    if best match in C is better match for this s than P_s
      P_s ← C_best
; P_s is cell in P responsible for recognizing
; this particular input pattern s

    generate a random population of n2 strings called R
; We assume than n2 ≤ N − |S| as the first |S| strings in P
; correspond to M and will not be replaced here

    replace "last" n2 strings in P with strings in R
  done
done

foreach(s ∈ S)
do
  M_s ← P_s
; output set M consists of "first" |S| strings in P

done
```

- Adaptation: Each antigen is presented to every antibody. The best (highest affinity) antibodies get a chance to clone, undergo somatic hypermutation, and potentially replace an existing memory cell;

- Decision-making: During training, the basic decision-making process is based on the current memory cells' and antibodies' reactions to the antigenic pattern (i.e., clonal selection). After training the decision of how to identify/recognize an incoming pattern is based on the affinity of the memory cells (in particular the one memory cell with the highest affinity) to the given test pattern.

5.1.2 Analysis of Serial CLONALG

In [32], the authors offer an analysis of CLONALG for pattern recognition. In this subsection, we duplicate some of their efforts and provide more detail. For our analysis of this algorithm, we use the following notation, let:

- M be the number of patterns to learn or training antigens and also the number of memory cells to evolve;

- L be the number of features in each cell;

- N be the number of antibodies in $\mathbf{P}$;

- $n1$ be the number of antibodies chosen for cloning;

- $n2$ be the number of antibodies to replace after each antigenic exposure;

- N_c be the number of clones produced;
- β be a clonal scalar factor; and
- G be the number of generations the algorithm is run.

For step 1 of the algorithm, we can assume that at worst case the generation of a random population of cells requires deciding a random value for each of the L features in the population. Therefore, to generate a random population of N antibodies will take $O(LN)$.

In step 2, we find the majority of the work of the algorithm. For each of the M antigens, we perform the following steps:

1. Calculating the affinity between the antigen and each of the N antibodies. Assuming that affinity calculation requires evaluating each feature, then this step is $O(LN)$.
2. Sorting the antibodies based on affinity takes $O(N\log_2 N)$
3. Creating and mutating clones based on the $n1$ best antibodies requires further explanation. To calculate the number of clones a given antibody is allowed to produce, we follow the methodology outlined in [32] and allow a given antibody to create a number of clones in proportion to its affinity. The antibody with the best affinity was allowed to produce $\beta * N$ clones, the second best was allowed to produce $\frac{\beta * N}{2}$ clones, the third best was allowed to produce $\frac{\beta * N}{3}$, and so on. That is, we assume that the antibodies were sorted in order from best to worst $1 \ldots N$, and the total number of clones is

as in [32]:

$$N_c = \sum_{i=1}^{n1} \text{round}\left(\frac{\beta * N}{i}\right) \tag{5.1.2}$$

The β in equation 5.1.2 is a clonal scalar factor which for all of our experiments we have set to 1 (following the example of [32]), and the round (.) function rounds the operand to the nearest integer. Temporarily ignoring this rounding factor, the largest number of clones will be produced if $n1 = N$, giving us:

$$N_c \leq \beta * N * \sum_{i=1}^{N} \left(\frac{1}{i}\right) \tag{5.1.3}$$

The summation in equation 5.1.3 is the calculation of the Nth harmonic number H_N which can be approximated asymptotically as $O(\ln N)$. If the constant β is always less than N, then we can give an asymptotic upper bound on N_c: $N_c = O(N \ln N)$. Again, if mutation requires manipulating each feature, then mutating these clones will give us $O(LN \ln N)$ for this step.

4. Finding the best mutated offspring requires calculating the affinity for each clone: again, $O(LN \ln N)$.

5. Finally, replacing the worst $n2$ cells in $\mathbf{P}$ with randomly created cells requires $O(Ln2)$. Again, by worst we mean those cells which least matched the given input pattern (i.e., those with the highest value according the equation

5.1.1). In the algorithm, there is a restriction that $n2$ cannot be larger than $N - M$; therefore N is an upper bound on the value of $n2$, and this step can be bounded by $O(LN)$

Each of these steps is performed M times giving us an upper bound on this part of this main loop of the algorithm of

$$O(MLN\log_2 N) \tag{5.1.4}$$

While this gives us a definite upper bound, as we will see later in this chapter, it may be more useful to think of this bound in terms of our three parameters: N, $n1$, $n2$. This then gives us a running time per generation of:

$$\begin{aligned} T(M) &= O(M(LN + N\log_2 N + LN\ln(n1) + Ln2)) \\ &= O(M(L(N + N\ln(n1) + n2) + N\log_2 N) \end{aligned} \tag{5.1.5}$$

Equations 5.1.4 and 5.1.5 give us bounds on the per generation cost. This leaves for us to estimate G, the number of generations. As we saw in section 3.5.3, there is no straight-forward way to estimate the number of generations needed to reach convergence. While we can borrow results from the GA literature to help determine the number of antibodies needed to reach a quality solution, this does not provide the tools to predict the number of generations needed [80, 65, 18, chap2, pp13-31]. Also, considering the more cooperative nature of CLONALG and the high, variable mutation rate employed, it is unclear how applicable theoretical

findings from the GA community will be for this AIS. In [32], the authors simply run the algorithm for a fixed number of generations. While this is a long-standing method within the evolutionary algorithm community, it does not necessarily guarantee convergence, which is what CLONALG for pattern recognition tasks is designed to achieve. Regardless of the number of generations, however, the overall runtime for the serial version of CLONALG will be:

$$T(M) = O(GM(L(N + N\ln(n1) + n2) + N\log_2 N) \tag{5.1.6}$$

$$= O(GMLN\log_2 N) \tag{5.1.7}$$

5.1.3 Parallel CLONALG

Since, essentially, there are no connections among the cells in the CLONALG algorithm (as there would be in a network model of the immune system), there is no real need for all of this process to occur on a single processor. The goal of the algorithm is simply to discover a set of memory cells that can recognize the input pattern. However, the way the algorithm is formulated, the development of a single memory cell can occur independently of the development of the other cells. To clarify this, let us reexamine the pseudocode of CLONALG presented in figure 5.1.1. As this algorithm is formulated the first $|S|$ cells in P correspond to the memory cell set M. Each element in M is responsible for recognizing only one particular input pattern from S. During each generation, each pattern in S is presented to the cells in P one by one. During each pattern presentation, the cells

in P are exposed to the particular pattern from S and then are allowed to clone and mutate based on this exposure. The clones are then exposed to the given input pattern. At this point, only the cell in P responsible for recognizing this particular input pattern is changed if the best clone has a better affinity for the input than the current cell in P. All other elements of M remain as they were before the presentation of this input pattern. Additionally, only the "last" (if we think if P as an array of cells) $|P| - |S|$ cells in P are eligible for random replacement at the end of each generation. This maintains the individual recognition properties of the cells in M. This is one reason we chose the CLONALG algorithm as a starting place for our investigations into parallelizing immune-inspired algorithms. For the parallelization of CLONALG, the input set is simply divided by the number of processes involved in the parallel job. Each process then evolves that number of memory cells, which are specific only to the subset of the input given to that process. Once this finishes, a root process gathers all of these memory cells together in order to present the final evolved pattern recognizing cells to the user.

From an analysis point of view, the parallelization would modify equations 5.1.6 and 5.1.7 to:

$$T_p(M) = T_o + O(\frac{M}{np}G(L(N + N\ln(n1) + n2) + N\log_2 N) \tag{5.1.8}$$

$$= T_o + O(\frac{M}{np}GLN\log_2 N) \tag{5.1.9}$$

where T_o is the overhead associated with parallelization (e.g., communication time, network setup, etc.) and np is the number of processors used. In reality, the runtime might not be so neat as implied by these equations, however. The reduction in the size of M at a given processor should have an impact on the number of generations needed for convergence. Additionally, it is unclear what, if any, qualitative contribution the reaction of the antibody pool to the entire antigenic set has on the the overall solution quality or the time to convergence. Regardless, based on equation 5.1.9, we still expect to see significant speedup through the use of multiple processors.

5.2 Verification Experiments

In this section we present a series of experiments based on the ones presented in [32]. We explore this initially to confirm that no qualitative difference is introduced by our parallelization scheme. We also want to determine how each of the user-tunable parameters for the algorithm affect the overall runtime behavior in our parallel version.

5.2.1 Experimental Design

For the experiments described below, the binary character recognition data set discussed in [32] were used. This data set consists of 8 binary patterns representing a 10x12 graphical representation of the Arabic digits 0-4, 6, 7, and 9. The length

of each pattern to be recognized, therefore, is 120. Figure 5.2.1 shows this original data set, and figure 5.2.2 show the memory cells generated using 8 processors.

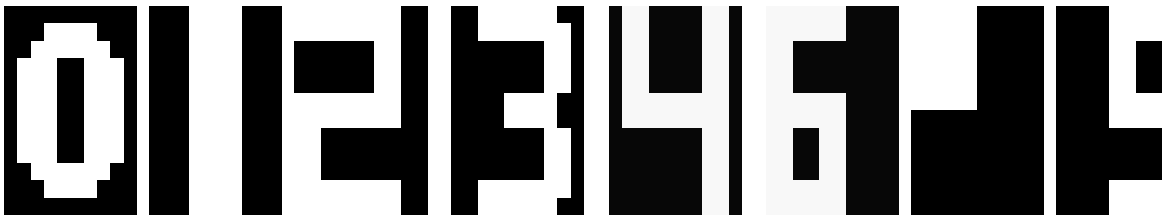

Figure 5.2.1: Original Input Data Set

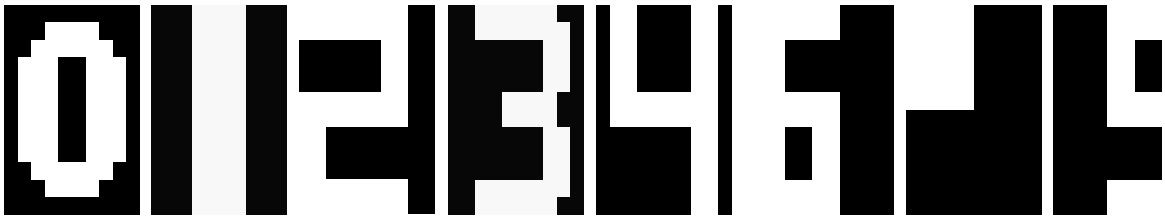

Figure 5.2.2: Memory Cells Generated

We investigate the effect three user-tunable–N, $n1$, and $n2$–parameters have on the behavior of the algorithm. We examine the overall runtime of the algorithm as well as the number of generations the algorithm takes to converge (where convergence is as described in 5.1). As mentioned in the previous section, CLONALG is almost embarrassingly parallel. In order to parallelize the algorithm, a root processor reads the input data in from a file, and then this input data was scattered across the processors, each of which then proceeded to use this divided data set as its input. These experiments were run on a cluster of dual-processor 2.4 Ghz Xeons using Gigabit Ethernet as the network fabric and the Message Passing Interface (MPI) [55] as the communication middleware with

one CLONALG process per processor. Once all of the processors completed, the resulting memory cells were gathered back to the root processor, and the overall runtime and the number of generations to convergence (max across the processors) were recorded. Results presented are the average and standard deviations from thirty runs.

5.2.2 Results

Figures 5.2.3 and 5.2.4 show the effects of varying the parameter N on the behavior of the algorithm both in terms of time to convergence and number of generations to convergence. This data is shown for runs across 1, 2, 4, and 8 processors. The timing graph is presented as log-log plot with 3σ error bars, while the generations graph is presented as a linear scale graph. Tables C.0.1 and C.0.2 present tables of these results with standard deviations in parentheses.

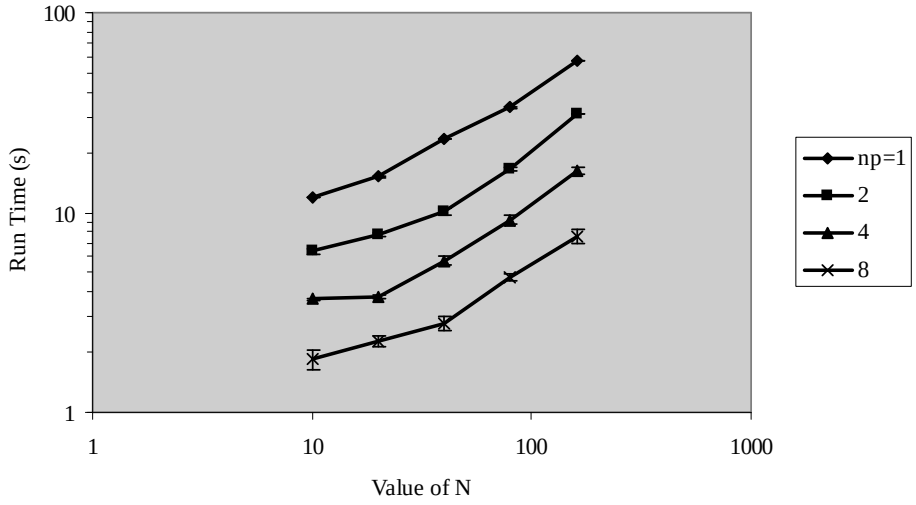

Figure 5.2.3: Effect of varying N parameter on Run Time

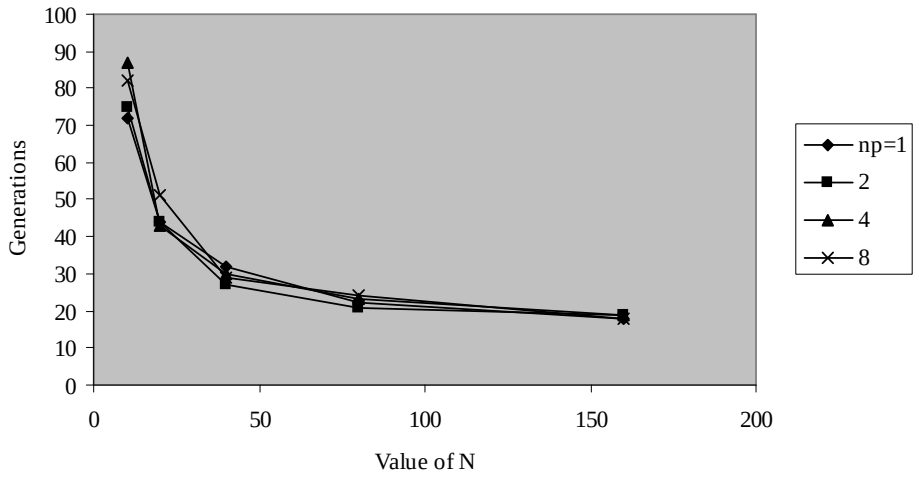

Figure 5.2.4: Effect of varying N parameter on Generations to Convergence

One thing these figures show is that the N parameter tends to have a fairly significant effect on the time and number of generations to convergence. What is interesting is that there seems to be a definite trade-off between time and number of generations. That is, the amount of overall runtime time increases as the value of N increases; however, the number of generations it took to reach this convergence seems to decrease as N increases. For example, in looking at tables C.0.1 and C.0.2, when the number of processors equals to four, we see the runtime increase with each doubling of N from 3.68 seconds when $N = 10$, 3.79 seconds for $N = 20$, 5.75 seconds for $N = 40$, 9.17 seconds for $N = 80$, and 16.2 seconds when $N = 160$. And the number of generations to convergence decreases from 87 when $N = 10$, 43 generations for $N = 20$, 30 generations for $N = 40$, 23 generations for $N = 80$, and 19 generations for $N = 160$. This increase in runtime roughly follows an $N\log_2 N$ pattern as predicted in equation 5.1.7, and this trend along with the

decreases in the number of generations to convergence is seen for the experiments when the number of processors was one, two, and eight, as well. Also of note is that while increasing the number of processors definitely decreases the overall running time of the algorithm, it has no real effect on the number of generations needed to converge on a solution. For example when $N = 160$ the runtime for one processor was 57.58 seconds, for two processors it was 31.08 seconds, for four processors it was 16.2 seconds, and for eight processors it was 7.64 seconds. However, for $N = 160$ the number of generations for convergence was 18 for one processor, 19 for two processors, 19 for four processors, and 18 for eight processors. This is not unexpected since the value of N is the same at each processor no matter how many processors are being used. We would not expect any real difference here. Practically speaking, the decrease in run-time is more important to us than the actual number of generations the algorithm took to reach this convergence. There is something to be said for larger values of our parameters and the number of cycles required to reach convergence. This indicates that convergence is encouraged through the diversity of cells being processed which is increased as the value of N is increased. Nevertheless, what we are most interested with these experiments is that we did not change the fundamental behavior of the algorithm (as is evidenced from the same convergence behavior for the same parameter values), but that we do decrease our overall runtime.

Figures 5.2.5 and 5.2.6 show the effects of varying the parameter $n1$ on the behavior of the algorithm both in terms of time to convergence and number of generations to convergence. This data is shown for runs across 1, 2, 4, and 8

processors. Again, the timing graph is presented as log-log plot with 3σ error bars, while the generations graph is presented as a linear scale graph. Tables C.0.3 and C.0.4 present tables of these results with standard deviations in parentheses.

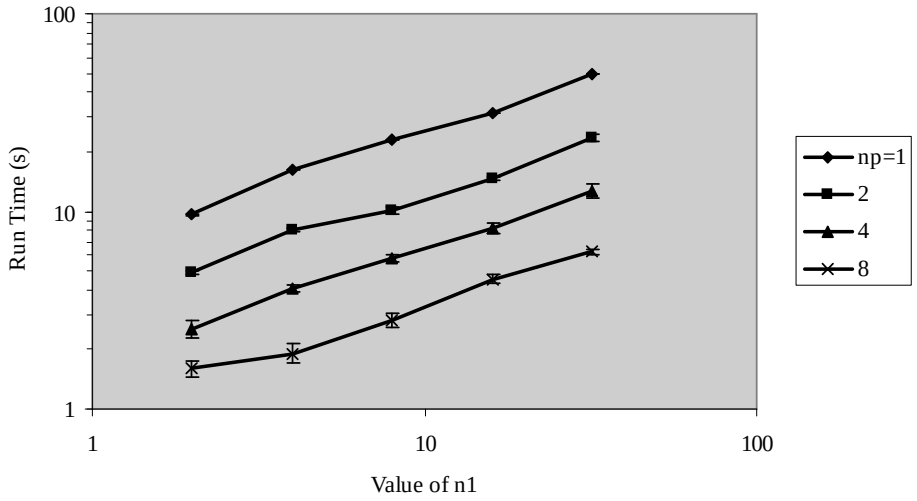

Figure 5.2.5: Effect of varying $n1$ parameter on Run Time

Again we see that an increase in the parameter's value leads to an increase in the running time of the algorithm, and again, we see that the number of processors employed decreases the running time. This is as predicted by equation 5.1.6 which indicates a growth in the runtime with a growth in the value of $n1$. However, it appears that the $n1$ parameter has little distinguishable effect on the number of generations needed for convergence. As in the previous experiments, we see that our parallel version has little impact on the convergence behavior of the algorithm which indicates a qualitative stability while providing a decrease in overall runtime.

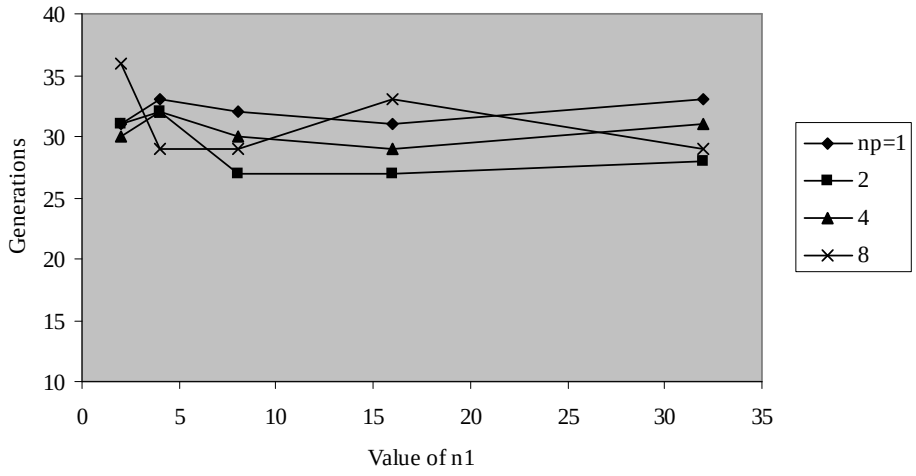

Figure 5.2.6: Effect of varying $n1$ parameter on Generations to Convergence

Figures 5.2.7 and 5.2.8 show the effects of varying the parameter $n2$ on the behavior of the algorithm both in terms of time to convergence and number of generations to convergence. This data is shown for runs across 1, 2, 4, and 8 processors. The timing graph is presented as linear-log plot with 3σ error bars, while the generations graph is presented as a linear scale graph. Tables C.0.5 and C.0.6 present tables of these results with standard deviations in parentheses.

The $n2$ parameter seems to have little overall effect on the behavior of the algorithm, whether in runtime or number of generations to convergence. Revisiting our analysis, we see in equation 5.1.6 that the $n2$ value has an extremely limited additive value to the run time behavior, and since $n2$ must always be on the order of N, it is unsurprising that there is little effect with this parameter. In general, these results indicate that for pattern recognition tasks the CLONALG algorithm seems insensitive to the $n2$ parameter.

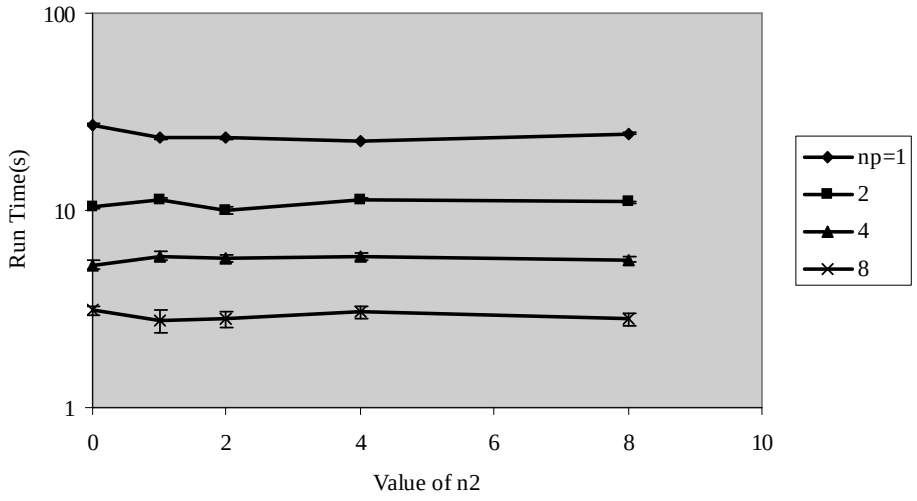

Figure 5.2.7: Effect of varying $n2$ parameter on Run Time

As mentioned in section 4.1, there are two fundamental metrics that we can use for assessing the gains of a parallel algorithm versus the serial version: speedup and parallel efficiency. Recall that speedup is defined as a ratio of the serial time over the parallel run time and can be thought of primarily as a performance metric. Ideally, speedup exhibits linear behavior such that with an increase in the number of processors one sees a proportional decrease in the run time. Efficiency, on the other hand, determines resource utilization with an ideal efficiency of 1 and is measured as defined in equation 4.1.3. Section B.2 discusses how these metrics were calculated for these experiments. Figure 5.2.9 shows the speedup obtained when varying the N parameter on these character recognition experiments presented in this section, and figure 5.2.10 shows the parallel efficiency for these same experiments. Tables C.0.7 and C.0.8 give the

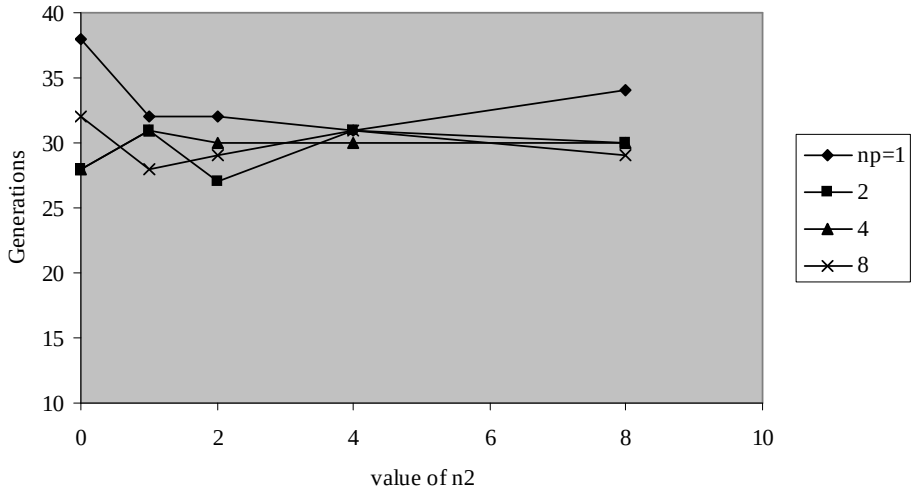

Figure 5.2.8: Effect of varying $n2$ parameter on Generations to Convergence

data tables for these figures. Figures C.0.1,C.0.2, C.0.3, and C.0.4 in appendix C provide these results for the $n1$ and $n2$ parameter variation.

In examining the efficiency curves we see that our 3σ error bars overlap on several data points. However, all of these efficiencies have a statistically significant difference ($n_1 = n_2 = 900, p < 0.01$; same value for N, different value for np). The efficiency results indicate that the parallel CLONALG is most efficient for this character recognition task when the value of N is 40 regardless of how many processors are used. Still, we find that when $np = 2$ and the value of N is 40 the system is able to most efficiently use these two processors.

With our speedup results, we see that with an increase in processors we get an increase in speedup as we would hope to see. In fact, we actually achieve better than ideal speedup or superlinear speedup on occasion. Looking at our efficiency curves we see this is also reflected in the greater than 1.0 efficiency values.

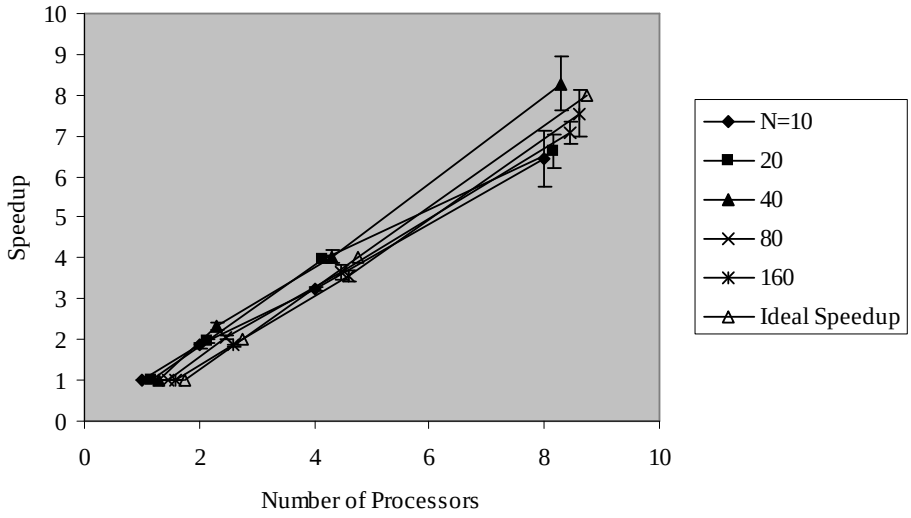

Figure 5.2.9: Speedup of CLONALG when varying the N parameter (x-axis offset applied for visual clarity)

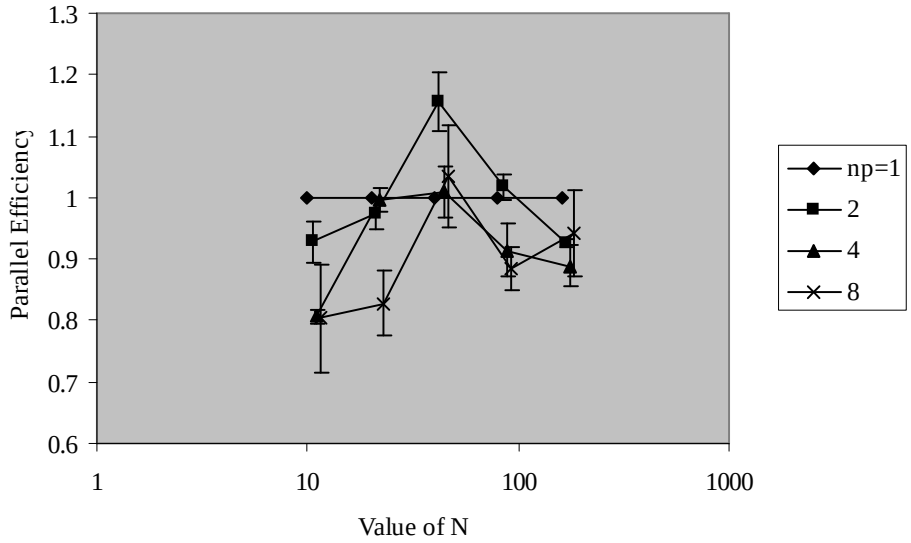

Figure 5.2.10: Parallel Efficiency of CLONALG when varying the N parameter (x-axis offset applied for visual clarity)

While this is extremely encouraging from a computational gain point of view, we should discuss why we may be seeing this "greater" than ideal performance. One reason is in the way the metrics are assessed. To be completely correct, speedup and efficiency should be measured against the most efficient serial version of an algorithm rather than just against the running of the parallel algorithm when $np = 1$. However, for these experiments, no fundamental change was made to the serial version, and running the parallel version with $np = 1$ is equivalent to running the serial version. The architecture of the system used for running these experiments may also have an influence as well. As mentioned earlier, these experiments were run on a cluster of dual-processor machines. For those processes run on the same machines (one on each of the two processors), it is likely that the MPI implementation took advantage of this and used the SMP capacities on these machines. This would lead to slightly more efficient communication from the process sharing a machine with the root process. Finally, there is really very little communication that occurs in the parallel version of this algorithm: the scattering of the training data to the various processors and a gathering of the developed memory cells at the end of training. This limited amount of communication means that the parallel overhead is minimal when compared to the rest of the algorithm.

5.2.3 Summary and Discussion of Verification Experiments

This section has presented results from our initial experiments with a parallelized immune-inspired learning algorithm. With any parallelization, it is important to asses what impact there is on the quality of the solution. This section used the data sets originally presented in [32] for this verification stage. As evidenced initially by figure 5.2.2, we found that our parallel version was giving us the same output as the serial version. Additionally, we found that varying the user-tunable parameters had similar effects on both the serial and parallel versions.

Satisfied with the qualitative consistency of the parallel version of CLONALG, we now turn our attention to the questions of performance. As previously stated, our primary goal in this study is to determine if parallelizing immune-inspired learning algorithms can provide computational gains. The results presented here indicate that there are definite gains to be had by parallelizing CLONALG. Both the speedup and efficiency results indicate near-ideal benefits from the use of multiple processors. Our next step is to examine the scalability of this parallel algorithm. For this, we will introduce simulated data sets that allow us to control the number of features and number of items used as input for parallel CLONALG.

5.3 Parallel CLONALG Scalability

To more fully asses the impact of our parallelization scheme on the behavior of CLONALG, we devised a series of "simulated" experiments on artificial data sets. These data sets are designed to provide full control on the number of feature (L) and the number of items (M) in each set. By varying these two characteristics about the data, we can begin to assess the scalability of this parallel system. Recall from section 4.1 that by scalability we mean the ability to maintain a given parallel efficiency while increasing the size of the input along with the number of processors used. We investigated the scalability of CLONALG in terms of both the number of input vectors and the number of input features.

5.3.1 Experimental Design

As we saw in section 5.2, the values of the user-tunable parameters have a definite impact on the overall performance of the algorithm. To limit this impact for our scalability tests, the value of N and $n1$ are kept constant at 160 for N (this is larger than any of the input sets tested) and 32 for $n1$. Since $n2$ seemed to have little impact on the behavior of the algorithm, we set it to 0. When assessing the scalability with respect to the number of input items, we generated data sets where the number of features was 64. When assessing the scalability with respect to the number of features, we generated datasets with the number of input items constant at 32. The following subsection presents the results from the scalability experiments. All results presented are an average of thirty runs.

5.3.2 Results

Figure 5.3.1 shows the effect of varying the number of input vectors on the run time of parallel CLONALG. This graph is presented as a log-log plot with 3σ error bars. Table C.0.13 presents these timing results in tabular form.[2] As predicted

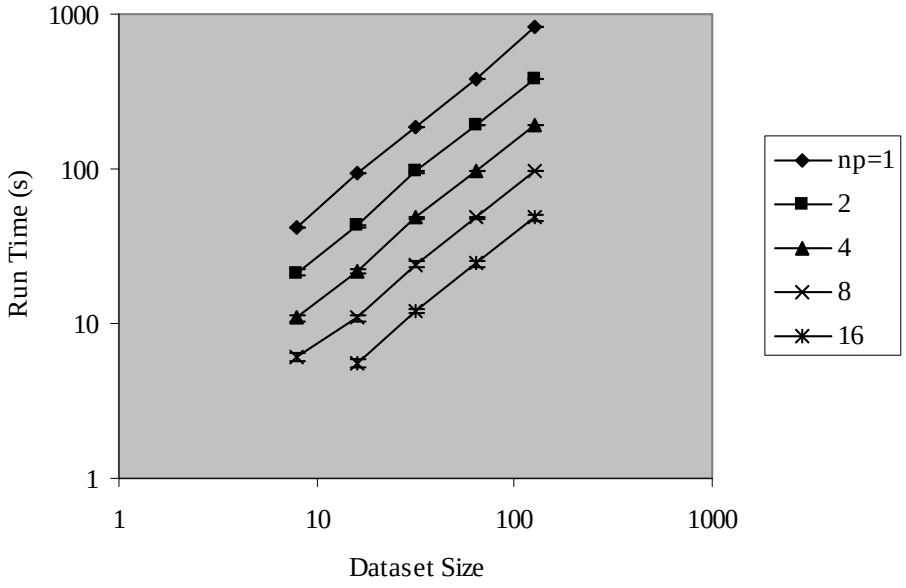

Figure 5.3.1: Parallel CLONALG: Effect of Varying the Number of Input Vectors on Run Time

by equation 5.1.7, an increase in the number of input items has a corresponding increase in the runtime. This is seen across the board for all number of processors used. However, we find that as more processors are used the overall run times decrease.

[2]Note that we only used up to M processors for any given set of experiments. This accounts for the "missing" values when $np = 16$ and $M = 8$ in these tables in this section.

Figure 5.3.2 presents parallel efficiency results as a log-linear plot with 3σ error bars, and tables C.0.14 and C.0.15 present the speedup and efficiency results when varying the number of input vectors presented to parallel CLONALG. For these

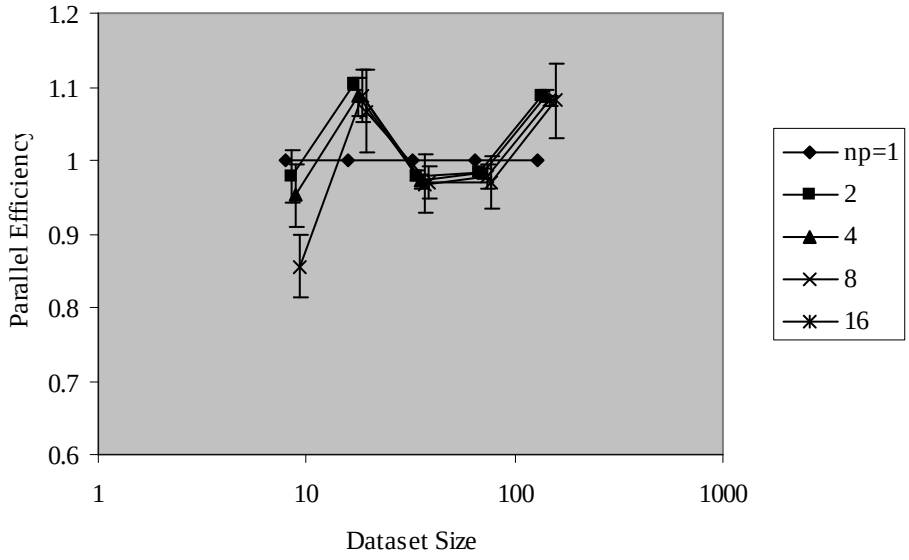

Figure 5.3.2: Parallel CLONALG: Parallel Efficiency when Varying the Number of Input Vectors(x-axis offset applied for visual clarity)

efficiency results, we find that for some values there is not a statsitically significant difference in the efficiencies ($n_1 = n_2 = 900, p \geq 0.01$; same value for M, different value for np). For example, when $M = 64$ the parallel efficiency is 0.98 when using 2, 4, and 8 processors. Given that parallel efficiency is a measure of how well the system is using the overall processing power available, there is no reason, necessarily, to expect there to be a difference for how well the parallel algorithm uses the processing resources.

Table C.0.14 also indicates the phenomenal performance gains to be had by increasing the number of processors used. The only limitation is that we cannot use more processors than we have input items (as seen when using 8 input items). Nevertheless, these results indicate that parallel CLONALG is definitely scalable in terms of number of input items.

Figure 5.3.3 shows the effect of varying the number of features on the run time of parallel CLONALG. This graph is presented as a log-log plot with 3σ error bars. Table C.0.16 presents these timing results in tabular form.

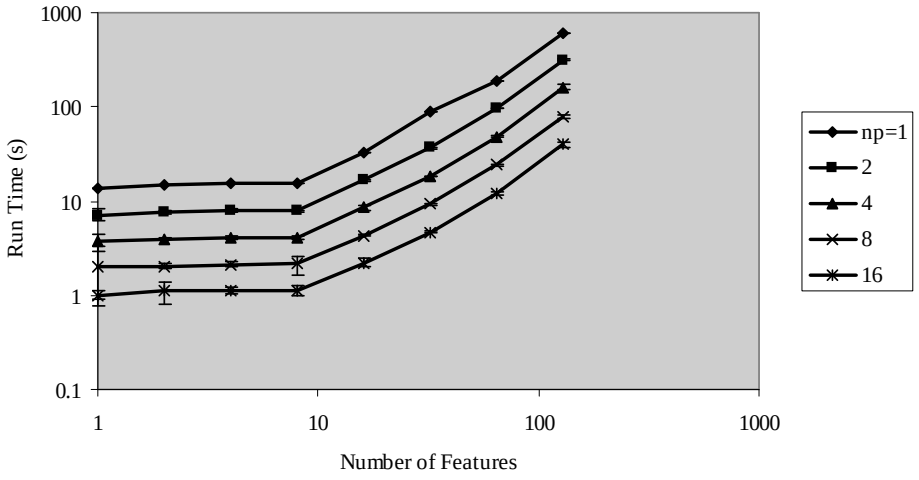

Figure 5.3.3: Parallel CLONALG: Effect of Varying the Length of the Input Vector on Run Time

Again, as predicted by equation 5.1.7, an increase in the number of features in each input vector results in an increase in runtime. Still, we once again find computational gains through the use of multiple processors. One item to note when examining figure 5.3.3 is that we find that for small number of features

($L \leq 8$) the rate of growth is more linear than the logarithmic growth predicted in our analysis.

Figure 5.3.4 presents parallel efficiency results as a log-linear plot with 3σ error bars, and tables C.0.17 and C.0.18 present the speedup and efficiency results when varying the number of features in each input vector on parallel CLONALG. Examining figure 5.3.4 we again find that, at times, there is little difference among

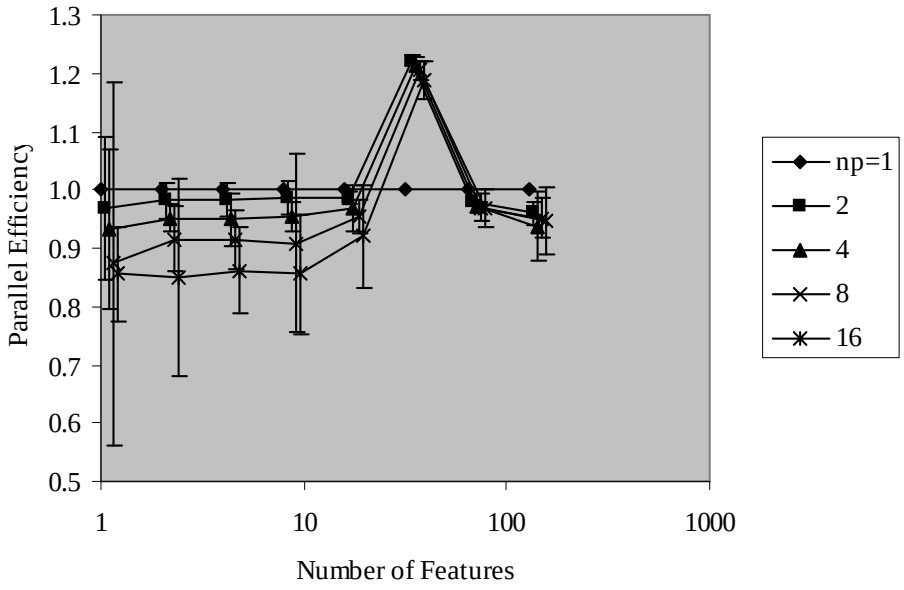

Figure 5.3.4: Parallel CLONALG: Parallel Efficiency when Varying the Length of the Input Vector (x-axis offset applied for visual clarity)

the parallel efficiencies across our sixteen processors. Interestingly, for all results there is a peak in accuracy when the number of features is 32. This may indicate an interesting behavioral characteristic of CLOANALG that it is most efficient with this number of features.

5.3.3 Summary and Discussion of Scalability Experiments

The basic scalability experiments presented in this section showed that the parallel version of CLONALG discussed here is definitely scalable in terms of both number of input items and the length of the input vector. We are able to maintain efficiency as we increase the number of processors while also increasing the input size. Again we see near ideal performance gains from utilizing more processors. The investigations, to date, do not suggest a limit in these performance gains; however, a system which is massively parallel with hundreds of processors may reveal a point where the communication times outstrip the parallel gains.

5.4 Summary and Concluding Remarks

This chapter has shown that even simple parallelization techniques can have a role in immune-inspired algorithms. Since often the goal of machine learning systems is to efficiently assist humans in the finding of interesting patterns in large amounts of data, any techniques that can speed up this process should have value. From our experiments with CLONALG, we were able to achieve a great deal of reduction in processing time through some basic parallelization. This is encouraging. Now that we have established that this application of standard computing technology will work for an immune-inspired algorithm, the next step in these parallelization experiments is the development of a parallel version of the AIRS algorithm.

Chapter 6

Parallel AIRS

Having examined in chapter 5 a technique for parallelizing an AIS algorithm, we now return to the AIRS algorithm introduced in chapter 3 as our primary example for the development of an immune-inspired learning algorithm.[1] Again, we are exploring the introduction of standard computational concepts in the overall development of our biologically-inspired algorithm. As with our discussion of CLONALG, we begin this chapter by examining those areas of the AIRS algorithm that can be readily parallelized. We discuss the key issues with our parallelization of AIRS that were not present in CLONALG. Then, we move on to perform verification experiments with parallel AIRS. Since AIRS is a classification algorithm, it is important that any changes we make to it do not adversely affect the classification accuracy. These experiments reveal other concerns that must be taken into consideration with parallel AIRS. We then examine the scalability of parallel AIRS through simulated study similar to section 5.3. As in the previous

[1] This chapter is an extended version of [129].

chapter with CLONALG, we explore the scalability of AIRS with respect to the number of features and the number of input vectors.

6.1 Parallelizing AIRS

We begin this section with a quick review of the serial version of AIRS. This is followed by several schemes for parallelizing this algorithm.

6.1.1 Overview of the AIRS Algorithm

As we saw in chapter 3, AIRS resembles CLONALG in the sense that both algorithms are concerned with developing a set of memory cells that give a representation of the learned environment. AIRS also employs affinity maturation and somatic hypermutation schemes that are similar to what is found in CLONALG. From AINE, AIRS has borrowed population control mechanisms and the concept of an abstract B-cell which represents a concentration of identical B-cells (referred to as Artificial Recognition Balls in previous papers). AIRS has also adopted from AINE the use of an affinity threshold for some learning mechanisms.

While we do not detail the entire algorithm here again,[2] we do want to highlight the key parts of AIRS that allows for an understanding of the parallelization. Like CLONALG, AIRS is concerned with the discovery or development of a set of memory cells that can encapsulate the training data. Basically, this is done in a two-stage process of first evolving a candidate memory cell and then determining

[2]See chapter 3 and appendix A for the pseudocode of AIRS.

if this candidate cell should be added to the overall pool of memory cells. This process can be outlined as follows:

1. Compare a training instance with all memory cells of the same class and find the memory cell with the best affinity for the training instance. We refer to this memory cell as mc_{match}.

2. Clone and mutate mc_{match} in proportion to its affinity to create a pool of abstract B-cells.

3. Calculate the affinity of each B-cell with the training instance.

4. Allocate resources to each B-cell based on its affinity.

5. Remove the weakest B-cells until the number of resources returns to a pre-set limit.

6. If the average affinity of the surviving B-cells is above a certain level, continue to step 7. Else, clone and mutate these surviving B-cells based on their affinity and return to step 3.

7. Choose the best B-cell as a candidate memory cell (mc_{cand}).

8. If the affinity of mc_{cand} for the training instance is better than the affinity of mc_{match}, then add mc_{cand} to the memory cell pool. If, in addition to this, the affinity between mc_{cand} and mc_{match} is within a certain threshold, then remove mc_{match} from the memory cell pool.

9. Repeat from step 1 until all training instances have been presented.

Once this training routine is complete, AIRS classifies instances using k-nearest neighbor with the developed set of memory cells.

6.1.2 Parallelizing AIRS

Having reviewed the serial version of AIRS, we turn our attention to our initial strategies for parallelizing this algorithm. Our primary motivation for these experiments is computational efficiency. We would like to employ mechanisms of harnessing the power of multiple processors applied to the same learning task rather than relying solely on a single processor. This ability will, in theory, allow us to apply AIRS to problem sets of a larger scale without sacrificing some of the appealing features of the algorithm.

Our initial approach to parallelizing this process is the same as the approach to parallelizing CLONALG presented in chapter 5: we partition the training data into np (number of processes) pieces and allow each of the processors to train on the separate portions of the training data. Figure 6.1.1 depicts this process. Unfortunately, unlike CLONALG which simply evolves one memory cell for each training data item, AIRS actually employs some degree of interaction between the candidate cells and the previously established memory cells. Partitioning the training data and allowing multiple copies of AIRS to run on these fractions of the data in essence creates np separate memory cell pools. It introduces a (possibly) significant difference in behavior from the serial version. So, when studying this parallelism, we must examine not only the computational efficiency

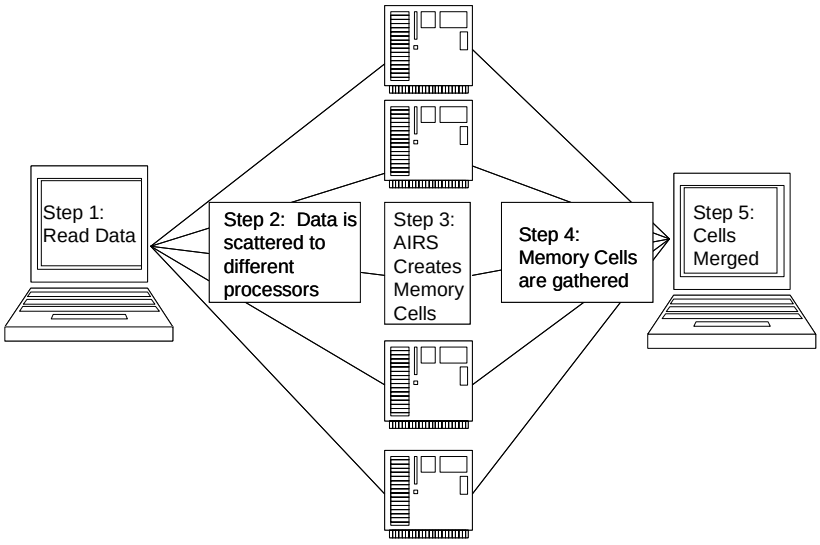

Figure 6.1.1: Overview of Parallel AIRS

we gain through this use of multiple processors, but we must also learn how evolving these memory cell pools in isolation of one another affects the overall performance of the algorithm.

Algorithmically, based on what is described in section 6.1.1, the parallel version behaves in the following manner:

1. Read in the training data at the root processor.

2. Scatter the training data to the NP processors.

3. Execute, on each processor, steps 1 through 9 from the serial version of the algorithm on the portion of the training data obtained.

4. Gather the developed memory cells from each processor back to the root processor.

5. Merge the gathered memory cells into a single memory cell pool for classification at the root processor. That is, the same processor used in step 1.

This parallel procedure forces us to explore a few design decisions. The primary difference between the parallel and serial version of AIRS are in steps 2 and 5. The scattering step (step 2) can be performed as we did with our parallelization of CLONALG. That is, we can randomly divide the training set evenly over the np processes and then allow each process to proceed as the serial version of the training algorithm. Since this method of parallelism creates np separate memory cell pools and since our classification is performed using a single memory cell pool, we must devise a method for merging the separate memory cell pools into one pool. As we persent in our verification experiments, there are several methods for handling this. It is with this method of scattering that we end up with the most issues concerning the merging step (step 5).

An alternative approach to the parallelization would be to split the data set in terms of class rather than random even partitions. With this method we could then remove the interaction that occurs and just dedicate a given process to evolving the memory cells for a particular class. This has the nice appeal of not requiring any method for compensating for the lack of global interaction that we see in our random distribution of training data. However, this limits the amount of processing power we can utilize to the number of classes in the data set. While for data sets with a large number of classes this limitation is insignificant, many

data sets require only differentiating between two classes. This implies that we could only utilize two processors. We examine this case for potential speedup as well, but recognize the inherent limitations. Still, this amount of speedup, while keeping the behavior of the algorithm the same as the serial version, may be well worth it.

In general, none of our parallel models change the fundamental behavior of the learning algorithm in terms of our MAD framework. At each individual processing site, the same model applies as follows:

- Memory: The memory of the AIRS algorithm is in the pool of memory cells developed through exposure to the training data (experiences);
- Adaptation: The adaptation occurs primarily in the ARB pool. With each new experience, AIRS evolves a candidate memory cell in reaction to this experience. If this memory cell is of sufficient quality, then the memory structure is adapted to include it.
- Decision-making: The initial decision is which one memory cell is most like the incoming training antigen. This cell is as a progenitor for a pool of evolving cells. During classification, the primary classification decision is based on the k most similar memory cells to the data item being classified.

Yet, while this model remains the same at each processing site, the merging techniques introduce a new, meta-learning component to the entire process. The memory of the algorithm remains the same; however, the development of this memory is altered based on how the memory cells are reassembled at the

root process. Each merging technique (with the exception of the class-based distribution) exhibits a slightly different model of memory development.

6.2 Verification Experiments

In this section we explore our methods of parallelization empirically by comparing the results obtained to the original serial versions. For these verification experiments, as with the experiments presented in section 5.2, we want to assess the impact our parallelization has on the behavior of the algorithm. Since AIRS is a classification algorithm, we are initially most concerned with maintaining classification accuracy. As we are exploring the scattering of the training data evenly over the np processors, we explore the development of different merging strategies. We then repeat these experiments but scatter the training data based on class. This method of scattering does not offer any changes in the behavioral characteristics of the algorithm and so does not require a special merging procedure.

6.2.1 Experimental Design

There are two merging strategies we explore here: concatenation and affinity-based merging. For the concatenation strategy, we simply allow each process to develop its set of memory cells, and then when these memory cells are gathered back to the root process, they are all added to the final memory cell pool. While this is an extremely naïve approach, we find that from a standpoint of computational gain

and classification accuracy, it works well. The second major merging strategy we examine is the affinity-based merging. For this strategy, we attempt to be more intelligent with our merging so that we do not have a large growth of memory cells in the final classifier, since, as we saw in chapter 3, one of the features of AIRS was its data reduction capabilities which was exhibited in the number of memory cells needed in the final classifier to represent the problem space. To this end, we utilize an affinity-based heuristic similar to what is used in the serial version of the algorithm for merging the memory cells.

For our class-based strategy, all data assigned to a given process is of the same class. When fewer processors are used than there are classes, then some processors are assigned more than one class. However, all of the training data of each class is kept intact. That is, it is never the case where the same class of data is scattered to multiple processors. This maintains the same behavioral characteristics of the serial version.

On a technical note, for the experiments presented in this section, we used the Iris, Pima Diabetes, and Sonar data sets that were used in previous studies of AIRS (chapter 3 and [130]). For all of these we took an average over 30 cross-validated runs and tested the parallel version on an increasing number of processors. In keeping with previous experiments on these data sets, we used a 5-fold cross-validation for the Iris data set, a 10-fold cross-validation for the Pima Diabetes data set, and a 13-fold cross validation for the Sonar data set. A cluster of dual-processor 2.4Ghz Xeons were used. The Message Passing Interface (MPI)[55, 106]

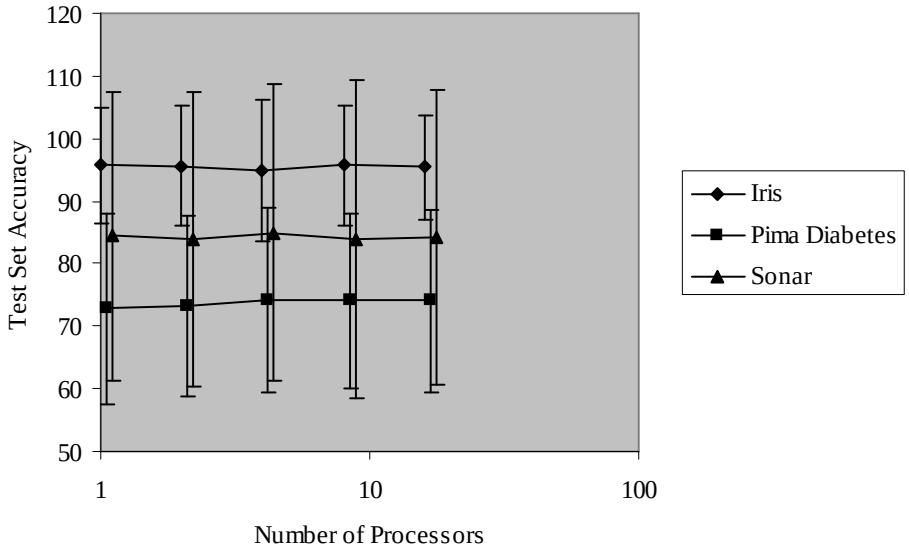

Figure 6.2.1: Parallel AIRS: Concatenation: Accuracies (x-axis offset applied for visual clarity)

was used as the communication library and communication took place over a Gigabit Ethernet network.

6.2.2 Results

Concatenation

Initially, we simply gathered each of the *np* memory cell pools at the root processor and concatenated these into a single large memory cell pool. As tables D.0.1, D.0.2, and D.0.3 and figures 6.2.1, 6.2.2, and 6.2.3 demonstrate, we were still able to achieve overall speedup in the process.[3] There are a couple of observations to be made from this initial set of experiments. Foremost, for our current purposes,

[3]Values in parentheses are standard deviations.

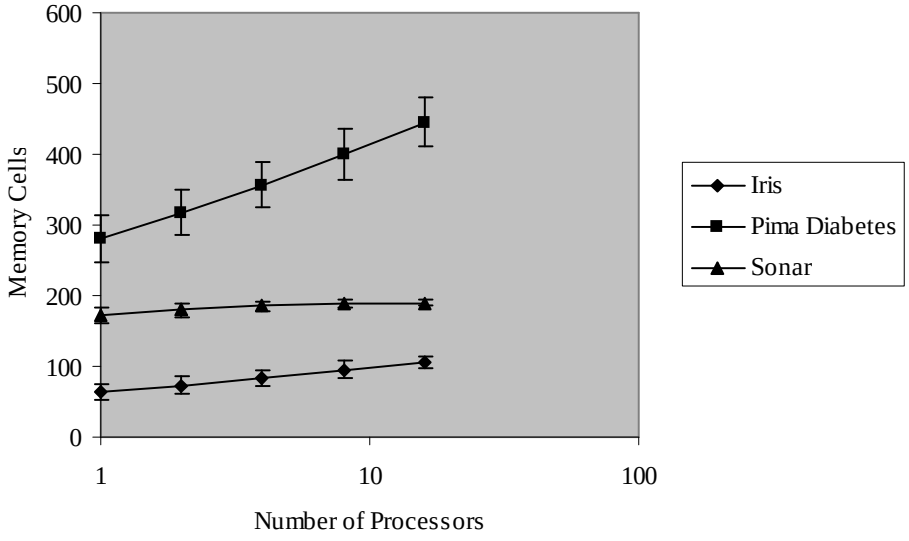

Figure 6.2.2: Parallel AIRS: Concatenation: Memory Cells

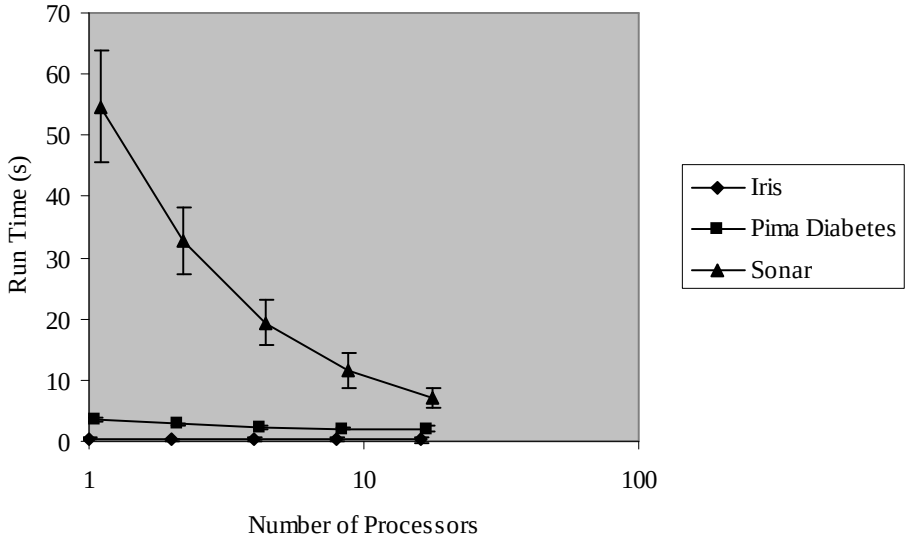

Figure 6.2.3: Parallel AIRS: Concatenation: Run Times (x-axis offset applied for visual clarity)

there is no loss in classification accuracy through our parallelization; therefore, the most important behavioral characteristic of AIRS—its classification capabilities—remains unchanged or improved. Looking at the accurarcy results, we find that for the iris and for the sonar data sets there is no statistically significant difference in the accuracies when compared to the serial accuracy (iris: $n_1 = n_2 = 150$ $p \geq 0.01$; sonar: $n_1 = n_2 = 390$ $p \geq 0.01$).[4] We do find a statistically significant difference when comparing the pima diabetes accuracies obtained with 4, 8, and 16 processors to the serial version ($n_1 = n_2 = 300$ $p < 0.01$); however, the change is an increase rather than a decrease in accuracy. Additionally, there is a statistically significant decrease in the overall runtime of the algorithm for two of the three datasets. For the Diabetes data set the overall runtime decreases from 3.43 seconds when one processor is used to 1.96 seconds when using sixteen processors. With the Sonar data set, we find an even greater (proportionally) decrease in runtime from 54.74 seconds when using one processor to 6.99 seconds when using sixteen processors. The Iris data set, however, is somewhat anomalous to this general trend of decreased runtime. The results indicate that for this data set essentially no runtime gain is to be had through this parallelization. While there is a statistically significant ($p < 0.01$) difference when comparing the runtime for one processor to the times obtained for more than one processor, when comparing the runtimes obtained with more than one processor with each other we find no statistically signifcant difference in their values. More than likely this

[4]Running the concatenation version of merging on one processor is equivalent in formulation to the original AIRS2 algorithm.

is due to the relative small size of the data set (only 120 training and 30 test samples per execution) and the limited number of features in the data set (only four features per vector). Essentially, the overhead of parallelization counteracts any gain that could be achieved on such a limited data set. Figures 6.2.4 and 6.2.5 and table D.0.4 provide speedup and efficiency graphs for these experiments.

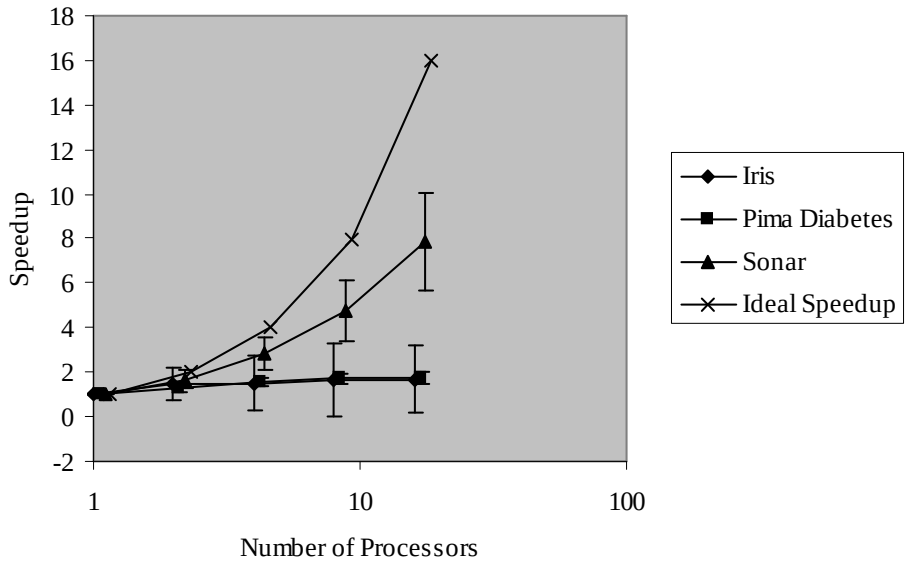

Figure 6.2.4: Speedup when Using Concatenation Merging Style (x-axis offset applied for visual clarity)

These curves are not as promising as the one seen in the previous chapter. However, these are much more typical of parallel algorithms: as we increase the number of processors we see a decrease in the efficient utilization of these processors. With the Diabetes and Sonar data sets we do continue to see a statistically significant rise in speedup with increased computational power, but not even this typical trend is exhibited with the Iris data set. With the Iris

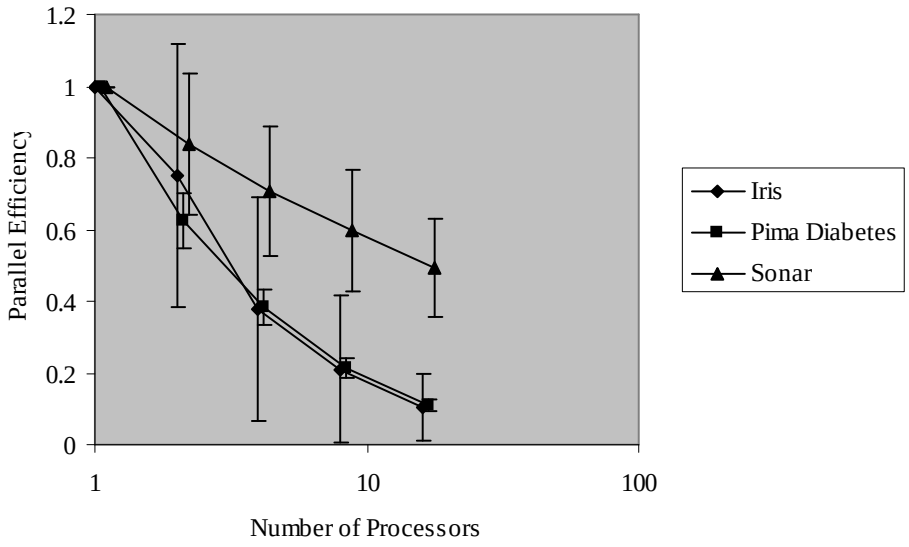

Figure 6.2.5: Parallel Efficiency when Using Concatenation Merging Style (x-axis offset applied for visual clarity)

data set, there is a statistically signifcant increase in speedup between one and two processors, and between four and eight processors; however, there is not a significant differences in the speedup values for two and four processors nor when comparing speedup for eight and sixteen processors. There is a statistically signficant decrease in the parallel efficiency values for all of these expereiments. For AIRS, we might initially assume that the more feature vectors in the training set, the greater the parallel efficiency. However, this is not the case. The Pima Diabetes data set has 691 training items in it; whereas, the Sonar data set has only 192 data items in the training set. Yet, examining the parallel efficiency results for these two data sets reveals that the Sonar data set has much more to gain from parallelization than does the Pima Diabetes data. The explanation for this

seeming discrepancy is in the number of features in each feature vector. The Pima Diabetes data has only eight features per feature vector, whereas the Sonar data has 60 (Iris has 4 features, incidentally). That our overall runtime (and its parallel efficiency) is predicated on the number of features in the data set should not be completely surprising. As with parallel GAs [18], the parallel version of AIRS is essentially dividing up the work of fitness (or affinity) evaluations. Additionally, we saw similar trends in section 5.3. For the current version of AIRS, affinity is determined based on Euclidean distance which is a metric whose evaluation grows linearly with the number of features. Thus, more gain is seen from data sets with both a large number of features and a large number of training instances when applying the parallel version of AIRS.

Affinity-Based Merging: Initial Approach

The results in the previous section exhibited another side-effect of note: with parallelization comes an increase in memory cells. As seen in tables D.0.1, D.0.2, and D.0.3, for the Iris data set this is an increase from 63.09 average number of memory cells for the serial version of the algorithm to 104.53 on average when using sixteen processors; for the Diabetes data set the increase is from 279.82 to 445.64; and for the Sonar data set the increase is from 173.1 to 189.9 when using sixteen processors. All of these increases are statistically significant at the 99% significance level when compared to the one process results. One of the hallmarks of AIRS has been its data reduction capabilities. As presented in [130], AIRS has been shown to reduce the amount of data needed to classify a given data set up

to 75%. This data reduction is measured in the number of memory cells present in the final classifier. To get an empirical sense of how the size of the memory cell set effects the classification time, we ran a set of experiments in which we compared AIRS to k-nearest neighbor (k-nn). Recall, that basic k-nn simply takes all of the training instances as examples and then classifies the test set through a majority voting scheme. AIRS first grows a set of memory cells which are then used to classify the test set. The results from these basic experiments on the Pima Diabetes data set are given in table 6.2.1 and provide a comparison between the number of training (Tr) and test cases used, the number of memory cells developed by AIRS (MC), and the difference in testing (T_{test}) and overall (T) runtime for k-nn and AIRS.

Table 6.2.1: Comparison of Runtimes for KNN and AIRS

Tr	Test	MC	T_{test}(KNN)	T_{test}(AIRS)	T(KNN)	T(AIRS)
692	77	277.850	0.149	0.072	1.290	3.521
615	153	254.040	0.276	0.115	1.181	3.048
512	256	217.433	0.373	0.154	1.012	2.468

Not surprisingly, when AIRS has greatly reduced the data set, there is a speed-up in time to classify the test set.[5] And, while not presented here, the accuracy of AIRS and k-nn have, for the data sets examined to date, always been roughly equivalent or AIRS has performed better (see, for example, table 3.3.1).

[5]In all fairness, it should be mentioned that the time to train in k-nn is virtually nothing, whereas the time to train in AIRS can be significant (when compared to 0). However, once the classifier is trained, it is the classification time that becomes most important as this is the task for which the classifier has been trained.

Since the classification speed of AIRS is based on the number of memory cells in the final pool, it is important to understand what impact parallelizing AIRS would have on the size of this set. In the serial version of AIRS, the minimum number of memory cells allowed is the number of classes that exist in the data set (one memory cell per class), and the maximum number is the number of data items in the training set, n, (one memory cell per training vector). For the parallel version, assuming that each process has examples of each class, the minimum at each process is the number of classes (nc); whereas, the maximum would be n/np. So, in the concatenation version of merging, the minimum number of memory cells in the final classifier increases from nc to $nc * np$. While one might suppose that the number of memory cells obtained through either the serial or parallel versions should be the same, it should be remembered that step 1 and (by implication) step 8 of the serial version depend on interaction with the entire memory cell pool. This interaction is not available in the current parallel version.

Our second approach to the merging stage is an attempt to minimize the number of memory cells that resulted from the pure concatenation approach. This method uses an affinity-based technique similar to step 8 in the serial version to reduce the size of the final memory cell pool. After gathering all the memory cells to the root process, they were then separated by class. Within each class grouping, a pairwise calculation of affinity between the memory cells was performed. If the affinity between two memory cells was less than the affinity threshold multiplied by the affinity threshold scalar, then only one of the memory cells was maintained

in the final pool. That is, if this relation:

$$\text{affinity}(mc_i, mc_j) < AT * ATS \tag{6.2.1}$$

(where mc_i and mc_j are two memory cells of the same class, the affinity threshold (AT) had been calculated across all of the training antigens as shown in equation 6.2.2, and the affinity threshold scalar (ATS) is set by the user) holds true, then mc_j is removed from the memory cell pool.

$$AT = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{affinity}(ag_i, ag_j)}{\frac{n(n-1)}{2}} \tag{6.2.2}$$

This merging technique was an initial attempt to compensate for the lack of global interaction the parallelizing process introduced. Tables D.0.5, D.0.6, and D.0.7 and figures 6.2.6, 6.2.7, and 6.2.8 give results when using this affinity-based merging. Keeping in mind that we are exploring these well-known data sets for verification, it is reassuring to initially note that this affinity-based merging technique has no adverse affect on the classification accuracy. While there is a statistically significant difference at the 99% significance level between the accuracies of the baseline version of AIRS (i.e., the concatenation merging style with one processor) and the accuracies on the Iris data set when using the affinity-based merging with two, four, and sixteen processors and on the Diabetes data set when using sixteen processors, these differences are relatively minor in terms

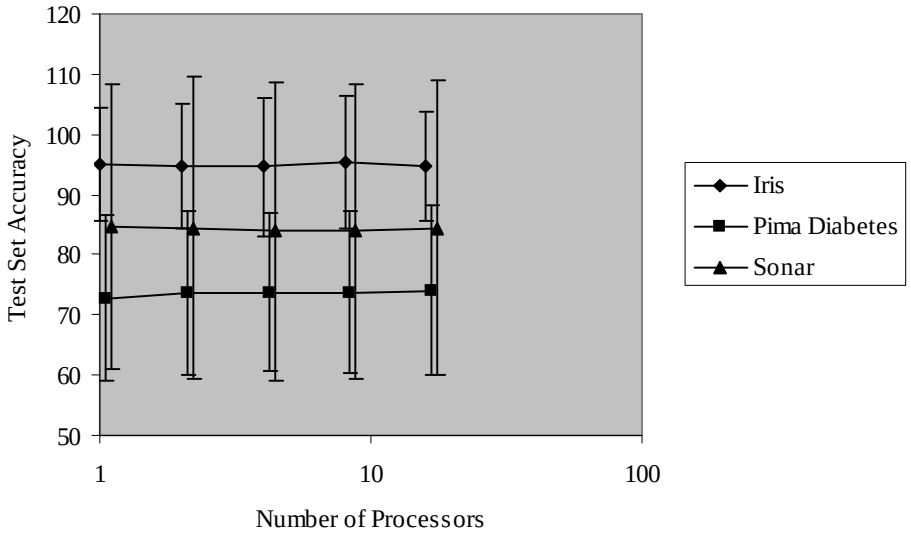

Figure 6.2.6: Parallel AIRS: Affinity-Based Merging: Accuracies (x-axis offset applied for visual clarity)

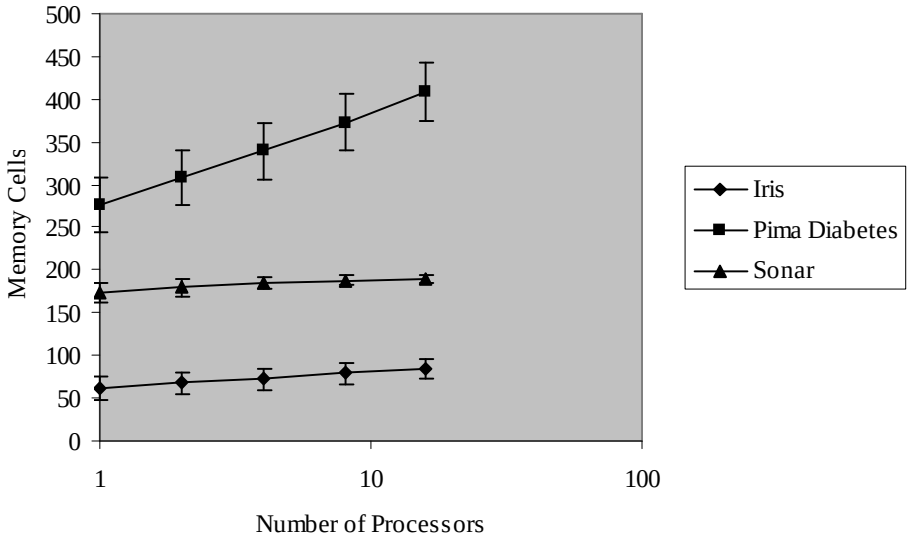

Figure 6.2.7: Parallel AIRS: Affinity-Based Merging: Memory Cells

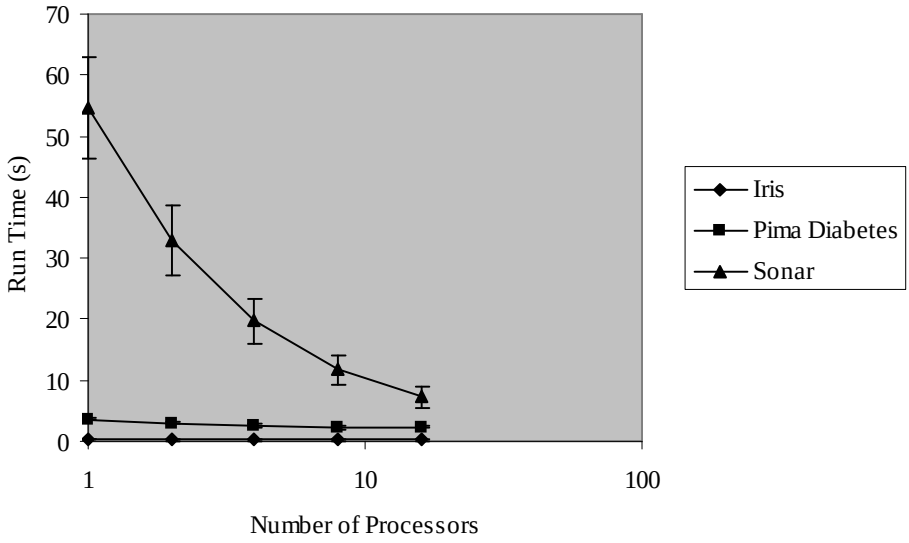

Figure 6.2.8: Parallel AIRS: Affinity-Based Merging: Run Times

of classification performance. Also, examining figures 6.2.9 and 6.2.10 and table D.0.8, we see similar parallel performance trends to the concatenation results.

In general, there is very little difference in the behavior of the algorithm on these data sets when using the affinity-based merging scheme when compared to the concatenation scheme. The two versions behave virtually the same. This includes the number of memory cells developed. Unfortunately, it appears that this attempt to maintain a constant memory cell size does not succeed. Since we are adding more overhead through this affinity based scheme, we would expect to see a slight drop-off in overall performance. Nevertheless, we still see gains through the use of multiple processors as evidenced, for example, by a decrease in runtime for the Diabetes dataset from 3.6 to 2.35 seconds when using one versus sixteen processors and for the Sonar dataset from 54.8 to 7.2 seconds.

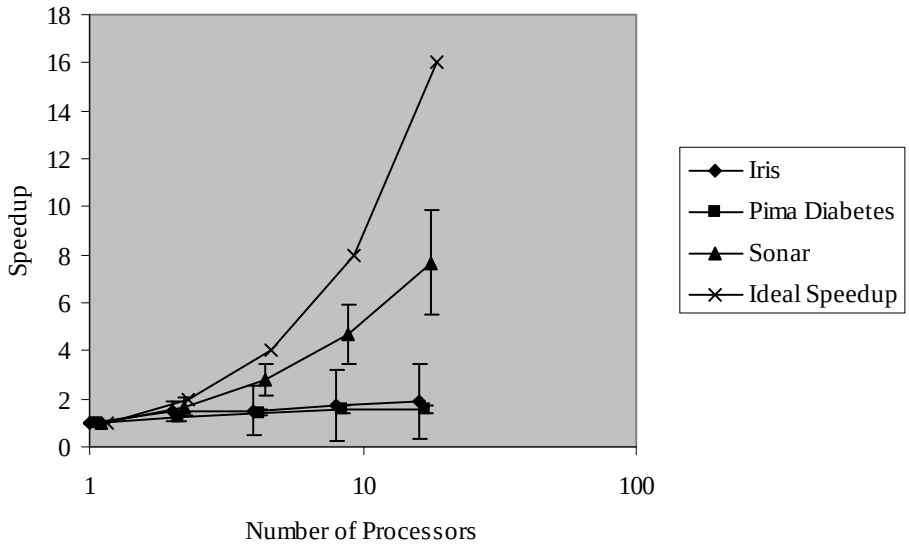

Figure 6.2.9: Speedup when Using Basic Affinity-Based Merging Style (x-axis offset applied for visual clarity)

Affinity-Based Merging: Revisited

As we just saw, the basic affinity-based merging technique employed did not significantly affect the increase in memory cells present in the final classifier. Clearly, the serial version of AIRS does not need as many memory cells to classify as accurately, so we would like to find a way to capture this further reduction in data while still employing our parallel techniques. Examining the increase in memory cells, there appears to be a roughly logarithmic increase with respect to an increase in the number of processors used. One method of remedying this increase in memory cells would be to alter the memory cell replacement criterion used in the affinity-based merging scheme by a logarithmic factor of the number of processors. That is, the criterion for removing a given memory cell is no longer

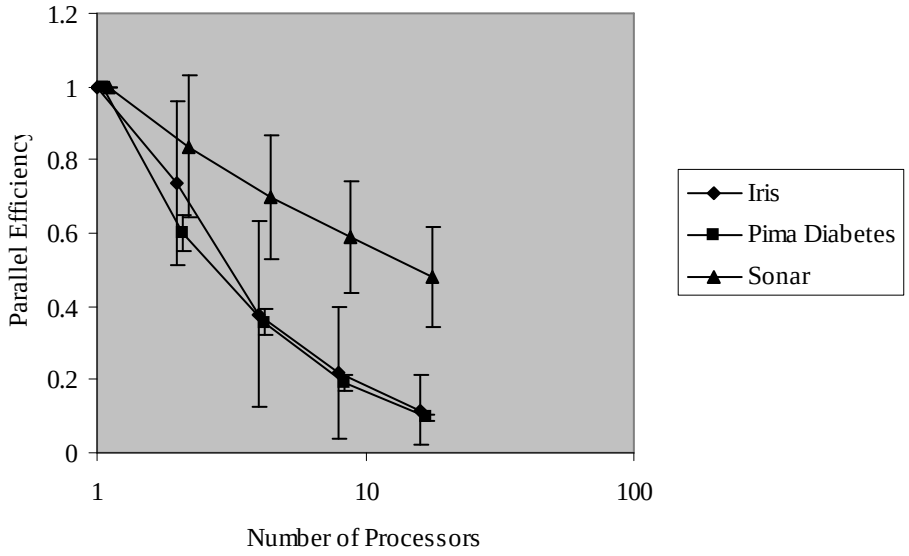

Figure 6.2.10: Parallel Efficiency when Using Basic Affinity-Based Merging Style (x-axis offset applied for visual clarity)

as specified in equation 6.2.1, but now the following relation must hold true for the removal of a memory cell:

$$\text{affinity}(mc_i, mc_j) < AT * ATS + \text{factor} \tag{6.2.3}$$

and factor is defined as:

$$\text{factor} = AT * ATS * \text{dampener} * log(np) \tag{6.2.4}$$

With the "dampener" referred to in equation 6.2.4 being a number between 0 and 1, this change to the merging scheme relaxes the criterion for memory cell removal in the affinity-based merging scheme by a small fraction in logarithmic

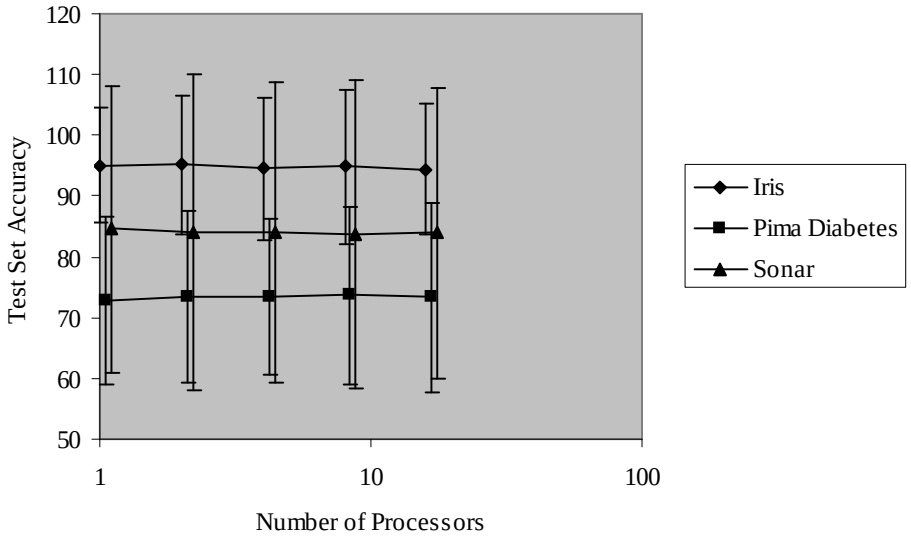

Figure 6.2.11: Parallel AIRS: Processor Dependent, Affinity-Based Merging: Accuracies (x-axis offset applied for visual clarity)

proportion of the number of processors used.[6] Tables D.0.9, D.0.10, and D.0.11 and figures 6.2.11, 6.2.12, and 6.2.13 below present results when employing this logarithmic factor to the criterion used in the affinity-based merging scheme.[7] The results from this new merging scheme are somewhat inconclusive. Again, we see no real impact on the classification accuracy. Our goal was to maintain the number of memory cells in the parallel classifier at a similar level to the serial version. However, this does not seem to be completely achieved. While the experiments on the Iris data set (table D.0.9) and the Pima Diabetes set (table D.0.10) do exhibit a reduction in the number of memory cells in the final classifier, it is unclear if this reduction would have continued unbounded if we had tested

[6]Obviously, the initial affinity-based merging scheme presented is just a variation on this new formulation with a "dampener" value of 0.

[7]An arbitrary value of 0.2 was used for the "dampener" value for these experiments.

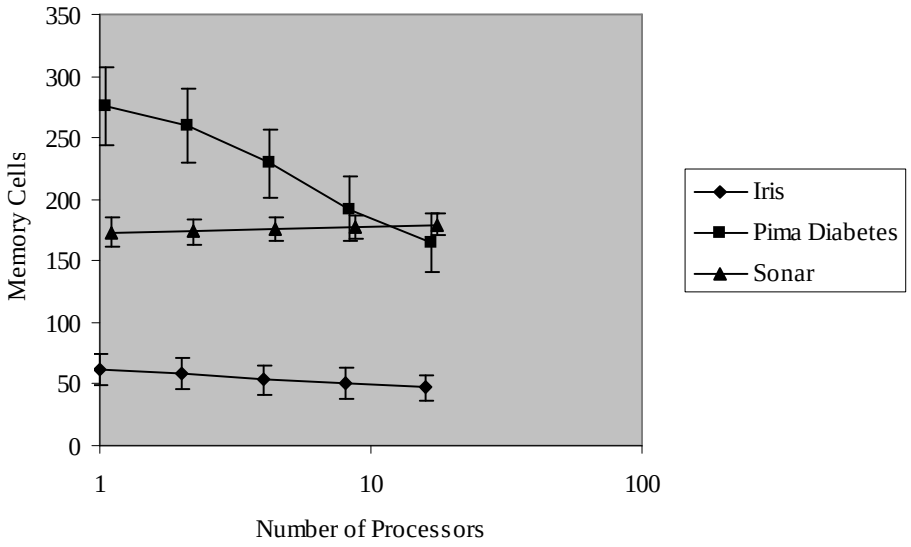

Figure 6.2.12: Parallel AIRS: Processor Dependent, Affinity-Based Merging: Memory Cells (x-axis offset applied for visual clarity)

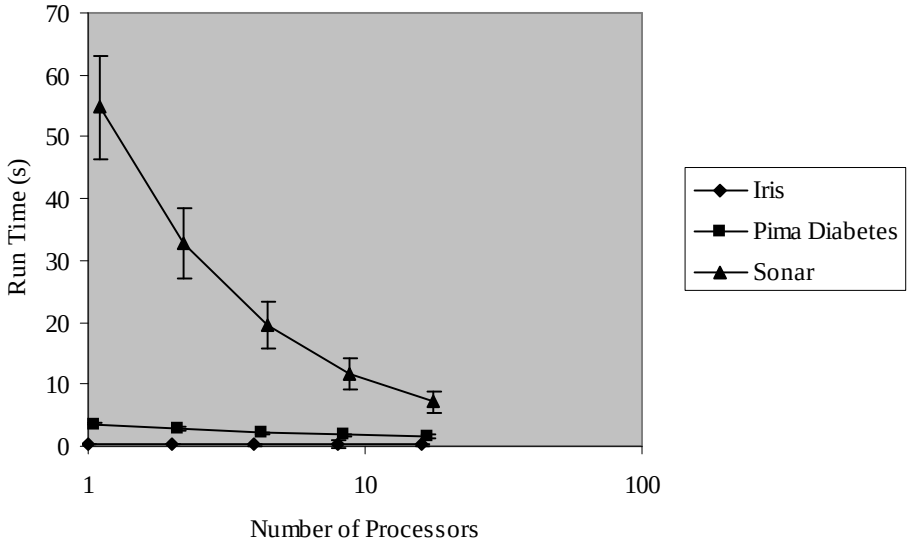

Figure 6.2.13: Parallel AIRS: Processor Dependent, Affinity-Based Merging: Run Times (x-axis offset applied for visual clarity)

on more and more processors. Eventually, with a significant decrease in memory cells, classification accuracy would decrease as well. On the other hand, with the Sonar data set experiments (table D.0.11), our new scheme appears to have achieved the basic goal of keeping the memory cells constant as we increase the number of processors. There is still a slight rise seen; however, it is not as extreme as in the previous experiments.

What all of this indicates may be simply that we have introduced another parameter (the "dampener" in equation 6.2.4) and that we need to determine the appropriate setting for this parameter for each classification task at hand. To assess the impact of this new parameter, then, we explored a range of values on these three benchmark data sets. Tables D.0.12 and D.0.13and figures 6.2.14 and 6.2.15 provide the test set accuracy and number of memory cells developed when varying this new parameter on the Iris data set, and table D.0.14 and figure 6.2.16 give the run times for this variation. Tables D.0.15, D.0.16, and D.0.17 and figures 6.2.17, 6.2.18, and 6.2.19 provide these results for the Pima Diabetes data set, and tables D.0.18, D.0.19, and D.0.20 and figures 6.2.20, 6.2.21, and 6.2.22 are these same results for the Sonar data set. Not surprisingly we see that an increase in the dampener value reduces the overall memory cell pool size. The Iris results (Table D.0.13) indicate that with a dampener value of 0.1 we seem to have achieved stability in the memory cell pool. We cannot definitively draw this conclusion as there is a slight increase in the number of memory cells with an increase in the number of processors particularly when comparing the runs with eight and sixteen processors to the run with one processor. The results for the

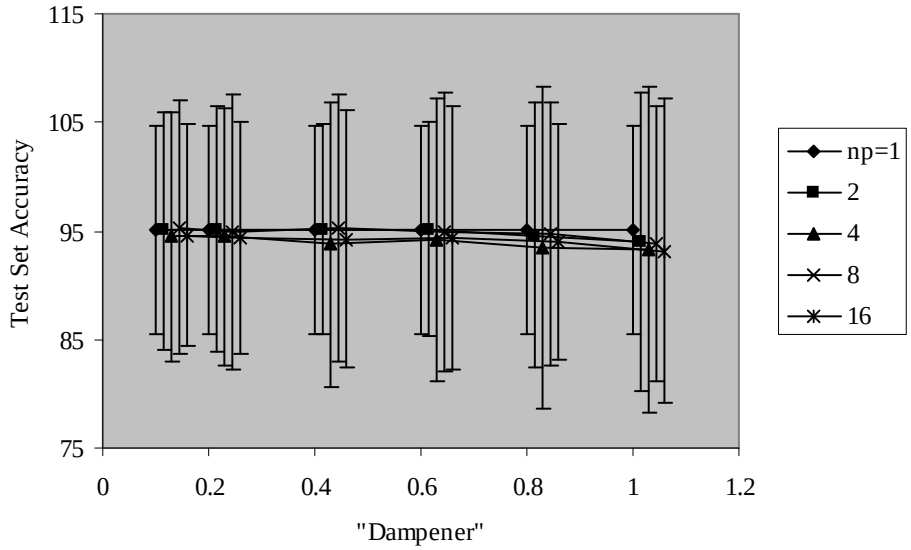

Figure 6.2.14: Iris Results: Varying the "Dampener": Accuracy (x-axis offset applied for visual clarity)

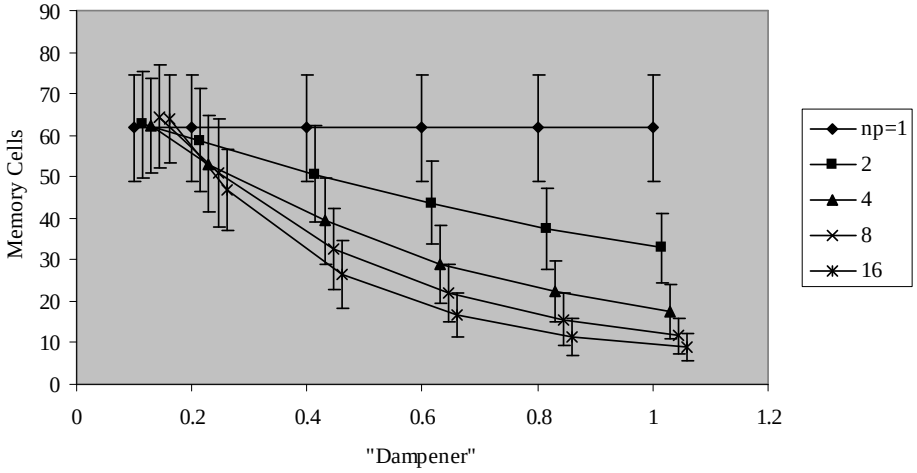

Figure 6.2.15: Iris Results: Varying the "Dampener": Memory Cells (x-axis offset applied for visual clarity)

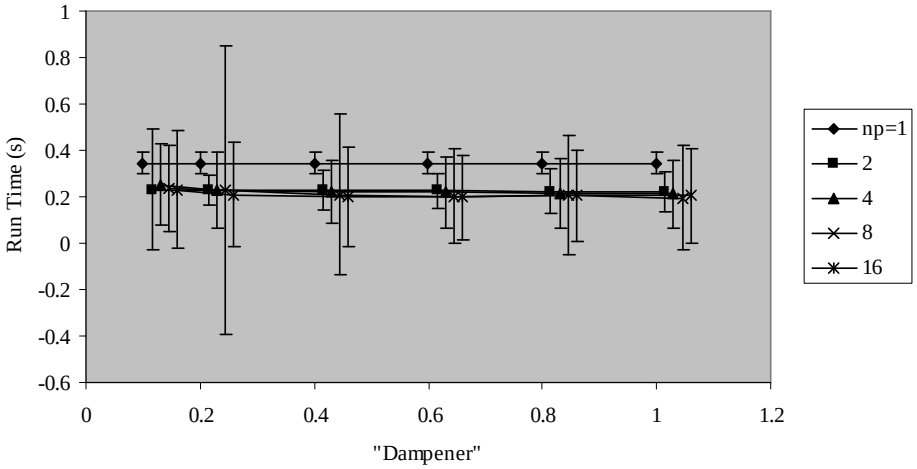

Figure 6.2.16: Iris Results: Varying the "Dampener": Run Time (x-axis offset applied for visual clarity)

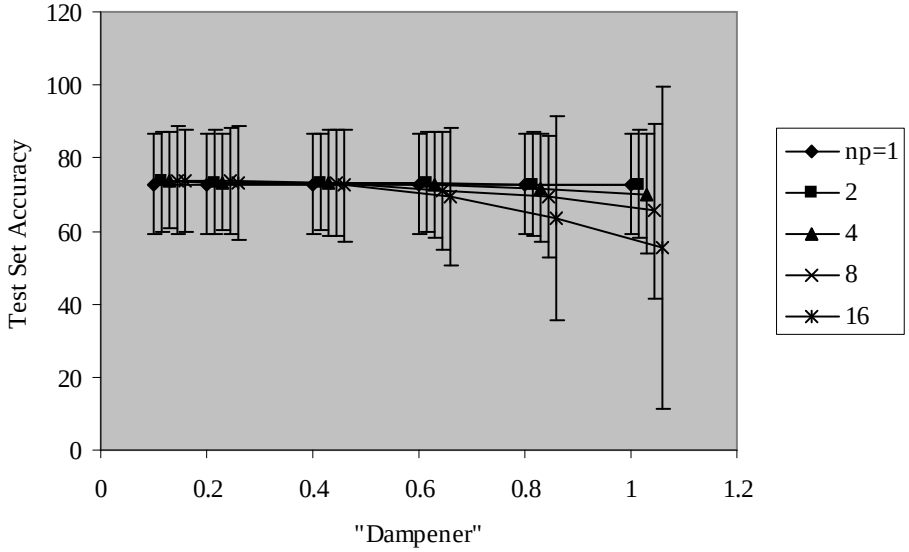

Figure 6.2.17: Pima Diabetes Results: Varying the "Dampener": Accuracy (x-axis offset applied for visual clarity)

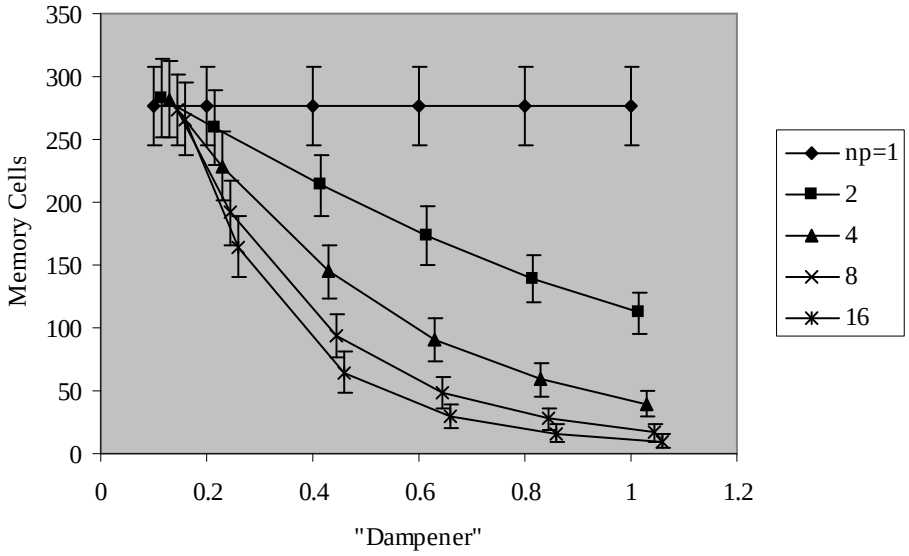

Figure 6.2.18: Pima Diabetes Results: Varying the "Dampener": Memory Cells (x-axis offset applied for visual clarity)

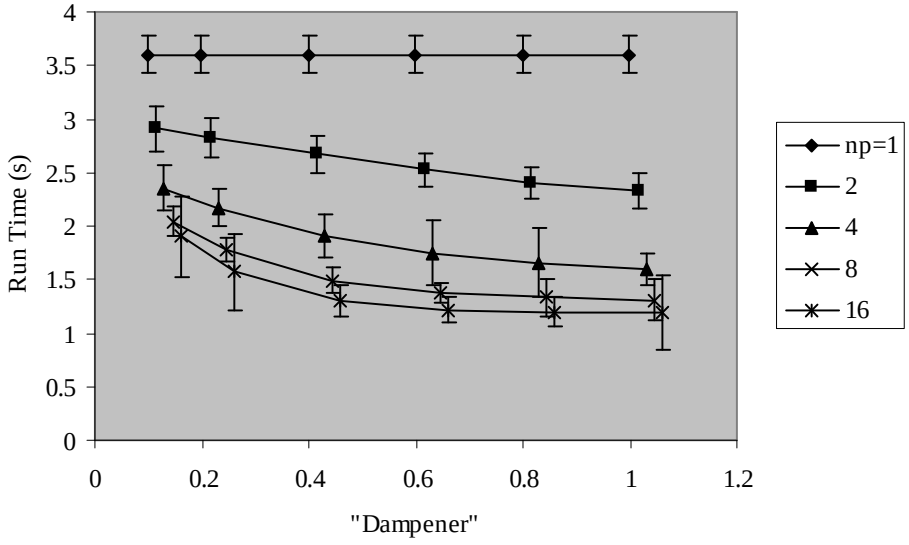

Figure 6.2.19: Pima Diabetes Results: Varying the "Dampener": Run Time (x-axis offset applied for visual clarity)

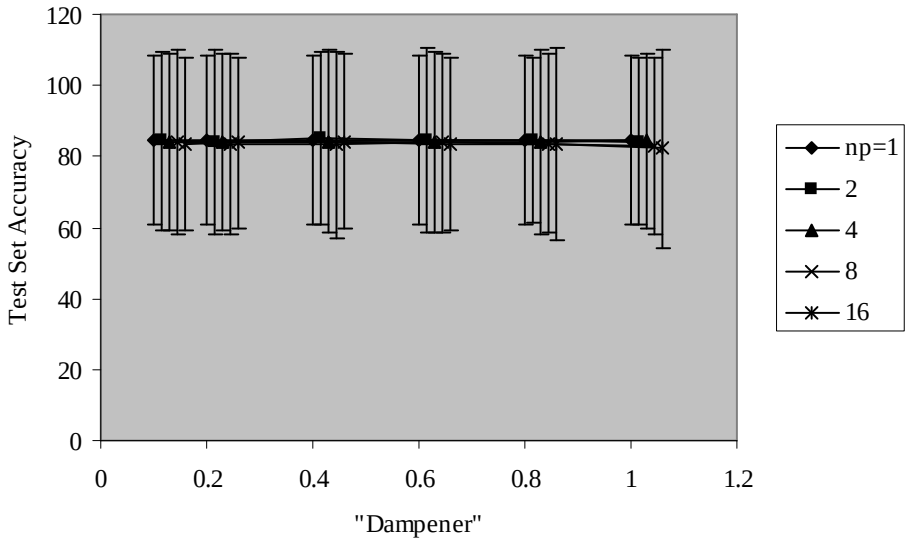

Figure 6.2.20: Sonar Results: Varying the "Dampener": Accuracy (x-axis offset applied for visual clarity)

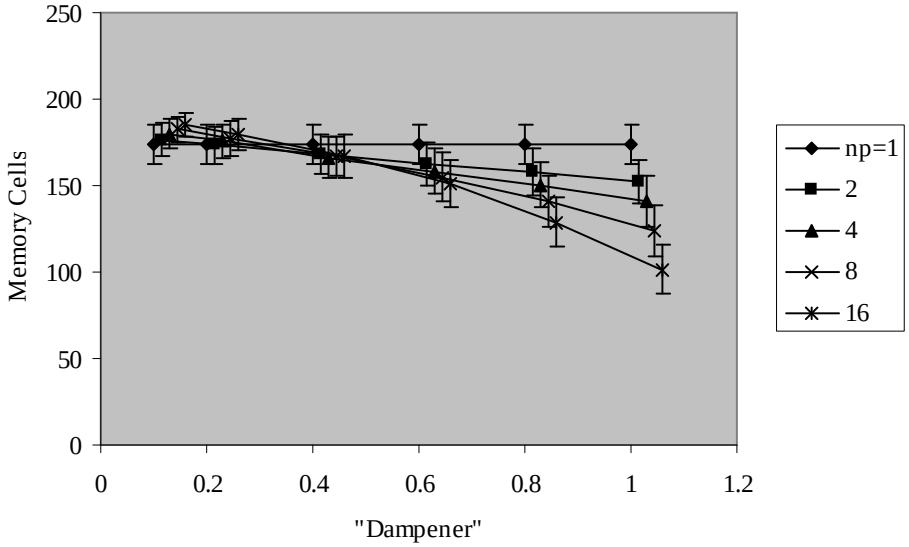

Figure 6.2.21: Sonar Results: Varying the "Dampener": Memory Cells (x-axis offset applied for visual clarity)

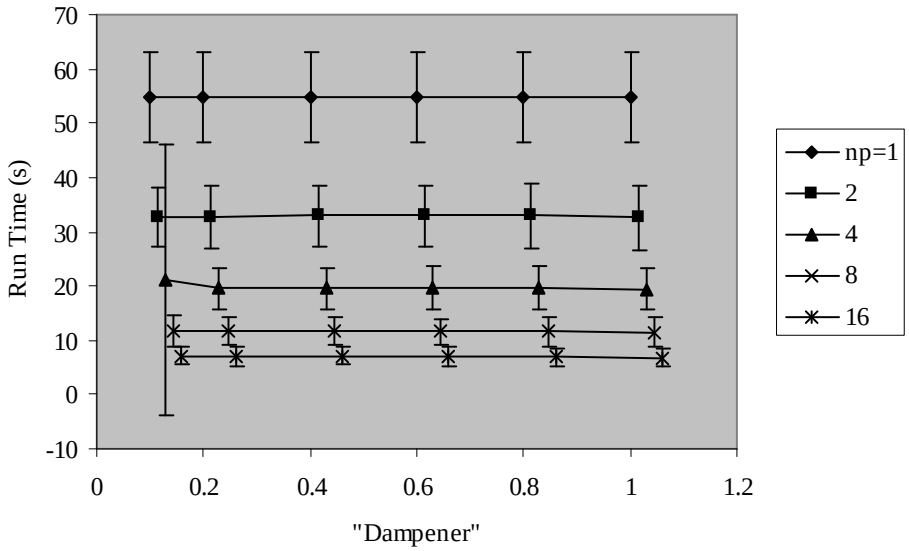

Figure 6.2.22: Sonar Results: Varying the "Dampener": Run Time (x-axis offset applied for visual clarity)

Pima Diabetes data sets (Table D.0.16) indicate that we have too large a value for this parameter. This can be deduced by the constant decrease, rather than a stability, in the memory cell pool size as we increase the number of processors. Looking at the number of memory cells for the Sonar data set (Table D.0.19) when the dampener parameter is set to 0.2, we find that we have achieved a somewhat more stable pool size, although an appropriate setting for this data set is probably somewhere between 0.2 and 0.4. Additionally, this stability is accompanied with no statistically significant impact on the predictive accuracy. Examining the columns in the timing results for these experiments (Tables D.0.14, D.0.17, and D.0.20) reveals that this decrease in overall memory cell pool size does not imply a significant decrease in overall runtime for the algorithm. Yet, the

reduction in the number of memory cells can have an impact on the classification accuracy as most prominently seen in the Diabetes experiments (Table D.0.15) when the "dampener" value is 0.8 or higher. Therefore, when using this version of merging, care must be taken in setting this dampener parameter. It could be argued that this method of merging does not provide a significant enough benefit (from a computational or memory efficiency point of view) to offset the potential for a decrease in classification ability. One positive from these results, however, can be seen in examining the timing results across the rows. For the Pima Diabetes and Sonar data sets we still see that increasing the number of processors does provide a reduction in the overall runtime of the algorithm. Once again, the use of these parallel techniques on the Iris data set seems to have no real computational gain.

Figures 6.2.23 and 6.2.24 and table D.0.21 present the parallel performance metrics for this processor dependent, affinity-based merging when using a dampening value of 0.2. We see similar trends in these figures as for the previous two merging schemes, with the Sonar data set seeming to have the most to gain from the parallelization of the algorithm as its speedup value increases from 1.68 when using two processors to 7.75 when using sixteen processors. These graphs again illustrate how the Iris data set receives no real benefit from these parallelization schemes as the speedup values remain low across the increasing number of processors.

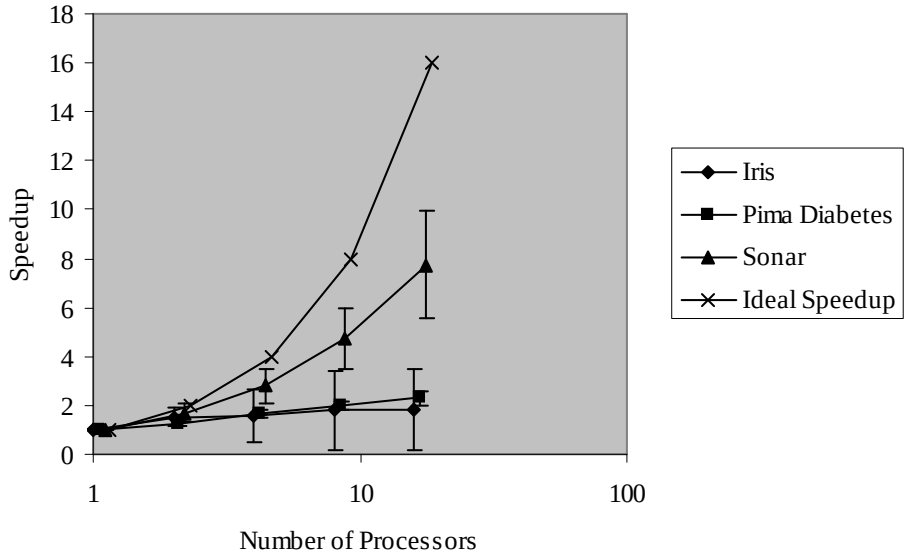

Figure 6.2.23: Speedup when Using Processor Dependent, Affinity-Based Merging Style (x-axis offset applied for visual clarity)

Class Parallelization

This final basic parallel strategy is based on the scatter stage rather than the gather and memory cell merging. For this method, the training data is scattered to processes based on class and each process is responsible for evolving memory cells specific to that class only. As previously mentioned, this limits the number of processes that are usable to the number of classes in the data set. Tables 6.2.2 present the runtime results when using this method of parallelization on the Iris (3-class data set), Pima Diabetes (2-class data set), and Sonar (2-class data set) data sets. Since the division of the training set occurs along class lines, no fundamental change in the classification or memory cell development compared to the serial version is expected. For all three data sets we find an increase in the

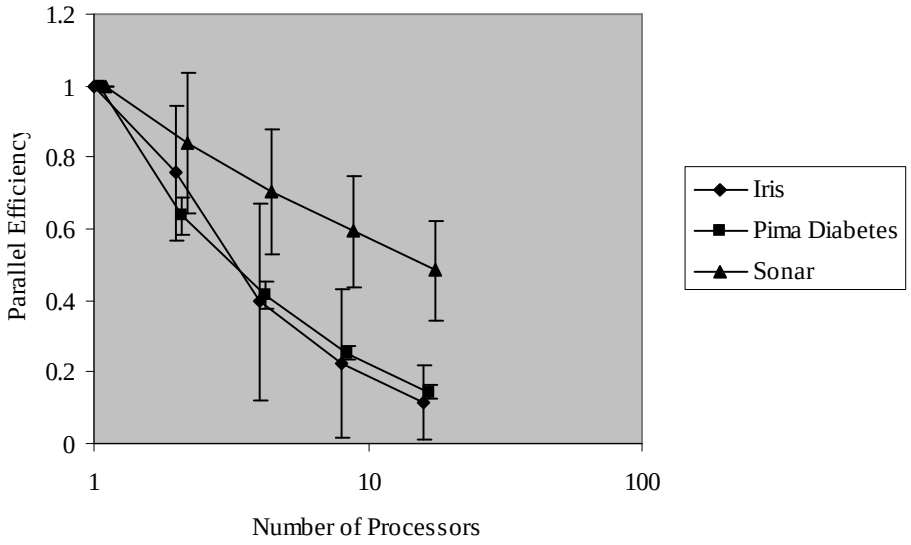

Figure 6.2.24: Parallel Efficiency when Using Processor Dependent, Affinity-Based Merging Style (x-axis offset applied for visual clarity)

Table 6.2.2: Run Times for Class Parallelization

np	Iris Run Time(s)	Pima Run Time (s)	Sonar Run Time (s)
1	0.34(0.05)	3.43(0.12)	54.74(3.03)
2	0.28(0.15)	3.03(0.06)	30.63(7.41)
3	0.23(0.06)		

number of processors produces a decrease in the overall run time, with the Sonar data set realizing the greatest benefit.

6.2.3 Summary and Discussion of Verification Experiments

This section has presented four different parallelization schemes for AIRS. The variation among the first three centers around the schemes for integrating the

memory cells developed at different processors into one memory cell pool useful for classification. With the first two merging schemes we saw little affect on the overall classification accuracy; however, there was a definite impact on the number of memory cells in the final classifier. Only the processor dependent, affinity-based merging scheme really addressed this issue. However, care must be taken when using this scheme to choose an appropriate value for the "dampener" parameter. The fourth scheme examined a different type of parallelization. Rather than trying to solve the memory cell integration problem posed by the random scattering of the data to individual processes, this final methods scattered the data along class lines. While this removes the serial dependencies of the other methods, it introduces inherent limitations. In the following section we examine this scheme for data sets that have more classes than the 2- and 3-class data sets explored in these verification experiments.

From a computational gains point of view, there seems to be no general conclusions we can make about the benefits of using parallel AIRS versus serial AIRS. For the Sonar data set, we see clear runtime gains through the use of multiple processors. For the other two data sets, we do find some gains, but these are much more modest. In our scalability experiments presented in section 6.3 we find that, in fact, our current models of parallelizing AIRS offer us very little in terms of stability of computational gains. In the next section, we see that AIRS does not appear to be scalable in terms of the number of training vectors nor in terms of the number of features in each vector. Nevertheless, despite this parallel system's instability, there are definite computational gains to be had. This was

evidenced with both the Sonar and Diabetes data sets in this section and is echoed in our scalability tests presented in the following section.

6.3 Parallel AIRS Scalability

To more fully assess the impact of our parallelization scheme on AIRS, we devised a series of "simulated" experiments on artificial data sets. These experiments are designed to echo the scalability investigations performed on our parallel version of CLONALG (section 5.3).

6.3.1 Experimental Design

To generate our datasets for the experiments varying the number and length of the input vectors, we utilized Powell Bendict's DGP-2 data generation program for inductive learning tasks [7]. This program is designed to produce synthetic data for testing learning algorithms. It allows the user to specify the number of features in each data instance, the amount of data, and the number of "peaks" (or centroids) to be used for the positive data examples. The program then generates synthetic data with both positive and negative examples based on these user parameters. Figure D.0.1 gives the parameter settings we used for these experiments. The only variation was, naturally, the number of features or the number of vectors produced. For the experiments varying the number of training vectors, we kept the number of features constant at 64. For the experiments varying the number of features, we maintained the number of training vectors at 256. We did an average

of 30 runs of 5-fold cross validation; so $n_1 = n_2 = 150$. The number of test data items was relative to the number of training vectors such that $T = \frac{N}{4}$, where T is the number of test data items and N is the number of training data items. All experiments utilized the processor dependent, affinity-based merging scheme with a dampener value of 0.1.

For the class division experiments, we used the data sets examined in [51]. These data sets consisted of feature vectors with two features and were created by dividing image files into class regions. Data sets with 3, 5, 8, and 12 classes were used.

6.3.2 Results

Varying the Number of Training Items

Table D.0.22 and figure 6.3.1 give the run times and table D.0.25 and figures 6.3.2 and 6.3.3 provides the speedup and parallel efficiency measures on the simulated data for parallel AIRS when varying the number of training instances.

Examining these results we see as discussed in section 3.5.3 that an increase in the number of training vectors leads to an increase in the overall runtime. Additionally, just investigating table D.0.22, we do find a decrease in overall runtime with the increase in processing power. These computational gains are dependent on the number of training vectors being used. For example, when only 32 training instances are available, there is virtually no benefit from the use of multiple processes. However, as the number of training instances increase, the

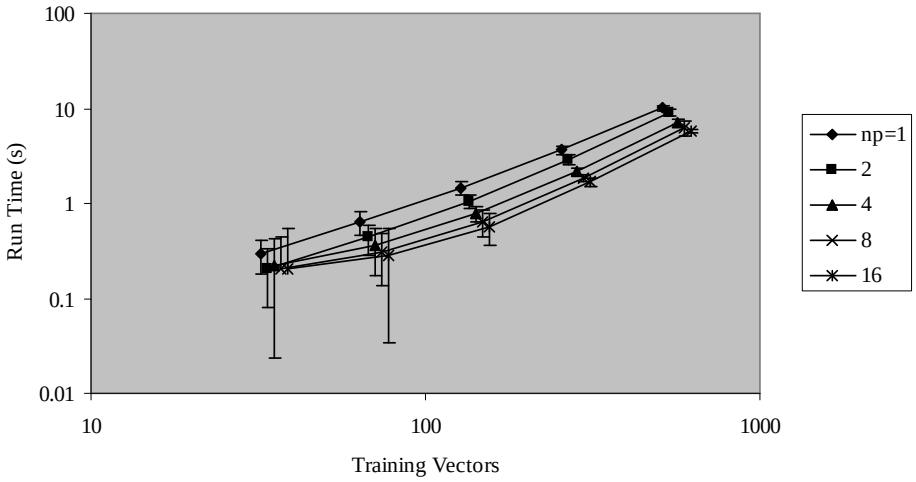

Figure 6.3.1: Parallel AIRS: Run Times when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

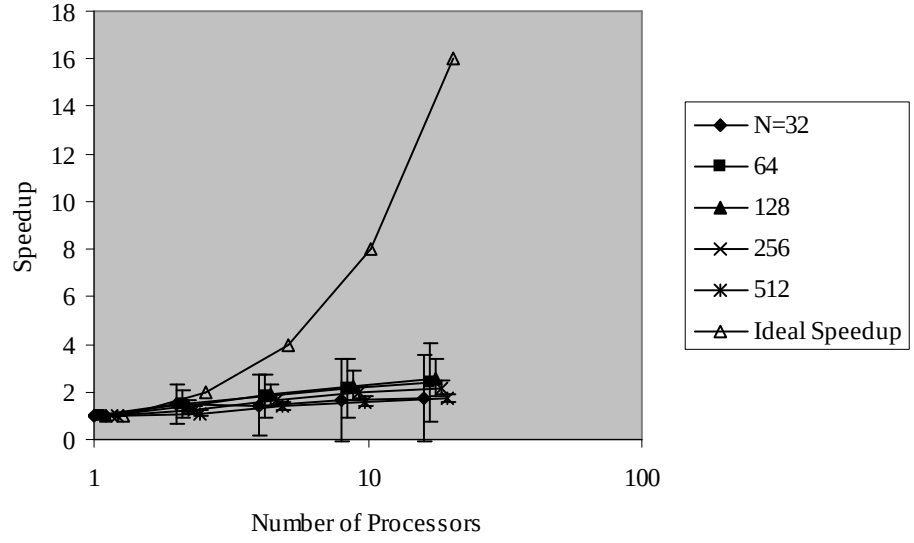

Figure 6.3.2: Parallel AIRS: Speedup when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

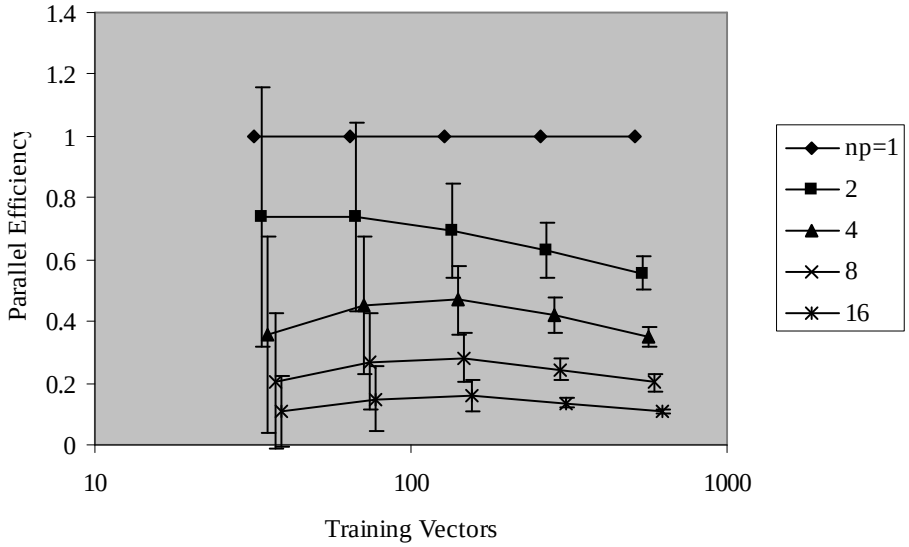

Figure 6.3.3: Parallel AIRS: Parallel Efficiency when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

benefits from parallelization also increase. This is the tale echoed in table D.0.25. While this table does indicate that this parallel version of AIRS is not scalable with respect to the number of training vectors, table D.0.22 still demonstrates clear gains when using the parallel version of AIRS. That is, even though we are not able to maintain a given level of efficiency with an increase in input size being accompanied by an increase in the number of processors, this potentially only impacts the predictability of this parallel system's performance and does not discount the potential time savings to be had.

While not the focus of these current experiments, tables D.0.23 and D.0.24 and figures 6.3.4 and 6.3.5 present the changes in the accuracy and number of memory cells developed when varying the number of input vectors.

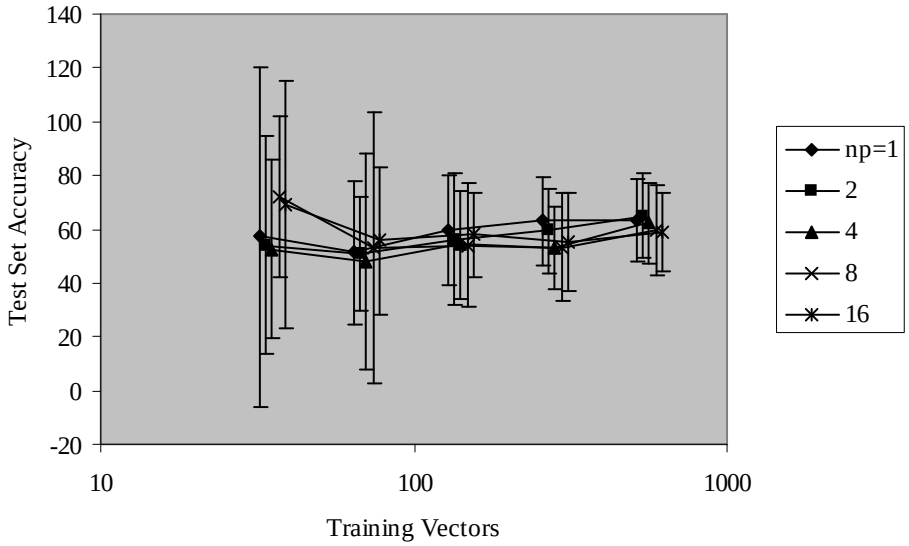

Figure 6.3.4: Parallel AIRS: Accuracy when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

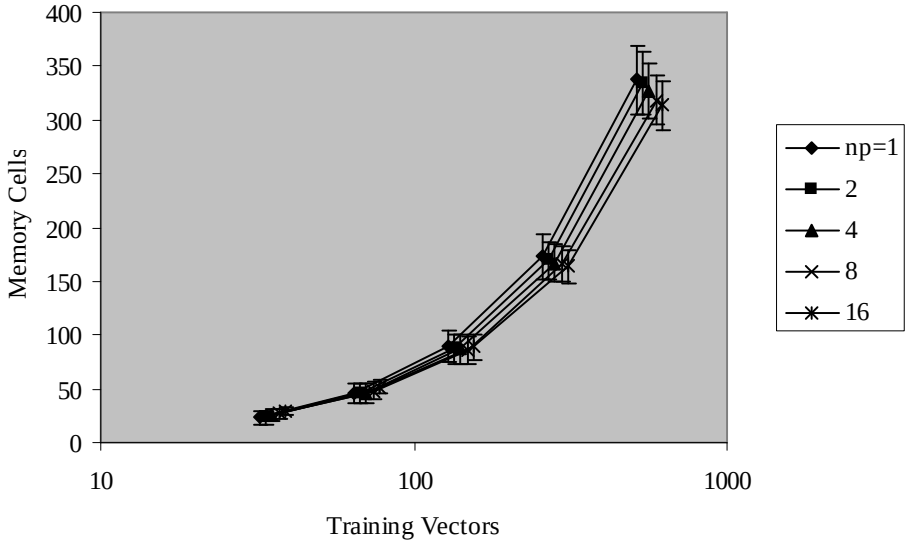

Figure 6.3.5: Parallel AIRS: Memory Cells when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

Not surprisingly, more memory cells are developed with more training items. However, there appears to be no real trend with regards to the accuracy rates and more training examples.

Varying the Number of Features

Table D.0.26 and figure 6.3.6 give the run times and table D.0.27 and figures 6.3.7 and 6.3.8 provide the speedup and parallel efficiency measures on the simulated data for parallel AIRS when varying the length of the input vector. Again, we

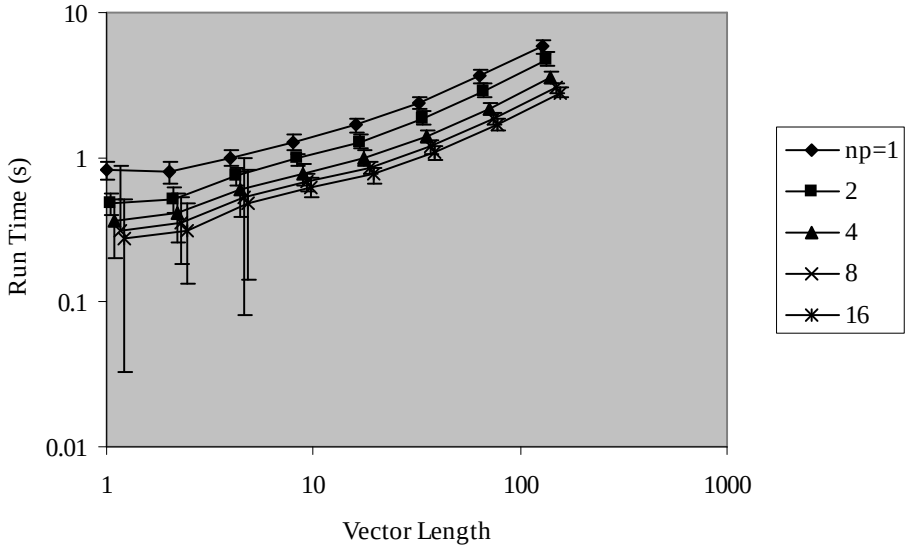

Figure 6.3.6: Parallel AIRS: Run Times when Varying the Length of the Input Vectors (x-axis offset applied for visual clarity)

see an increase in processing time with an increase in the number of features. While we do find a decrease in overall runtime with an increase in the number of processors, examining table D.0.27 reveals that this version of parallel AIRS does not appear to be scalable with respect to the number of features, either.

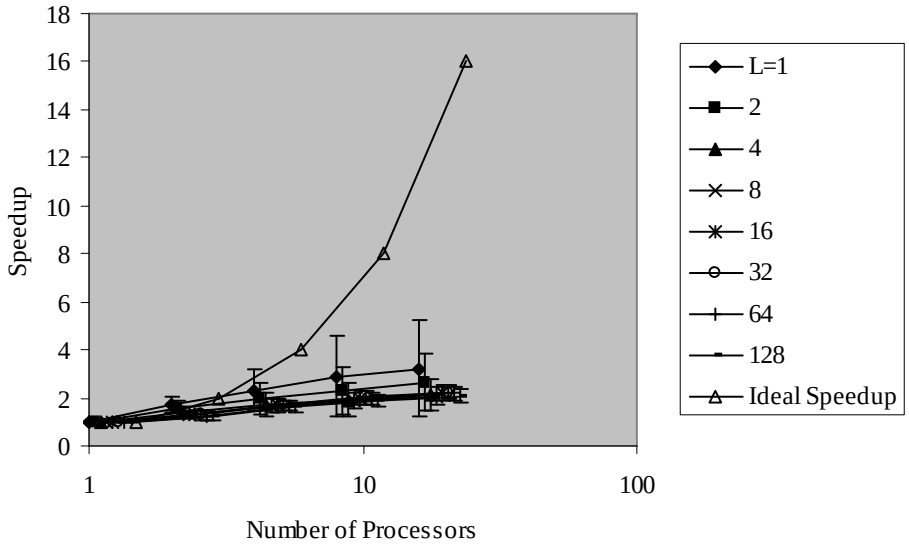

Figure 6.3.7: Parallel AIRS: Speedup when Varying the Length of the Input Vector (x-axis offset applied for visual clarity)

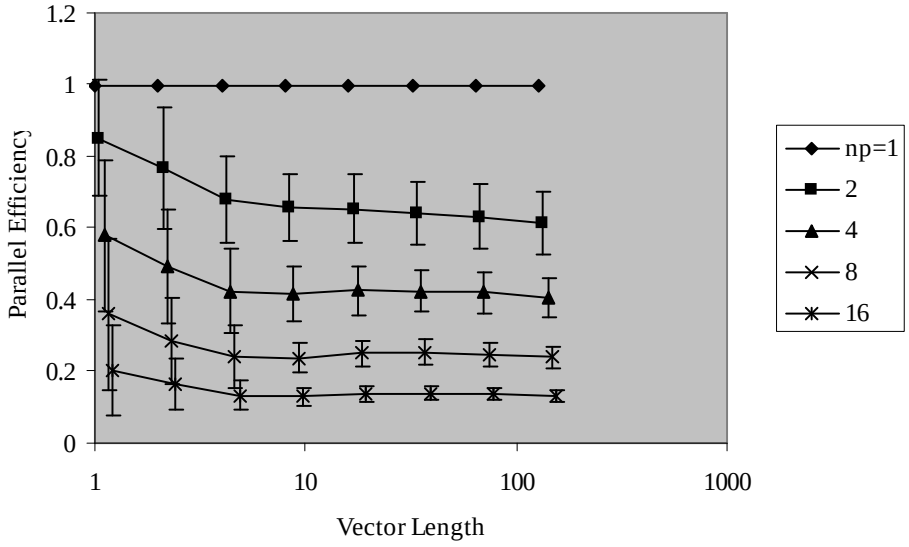

Figure 6.3.8: Parallel AIRS: Parallel Efficiency when Varying the Length of the Input Vector (x-axis offset applied for visual clarity)

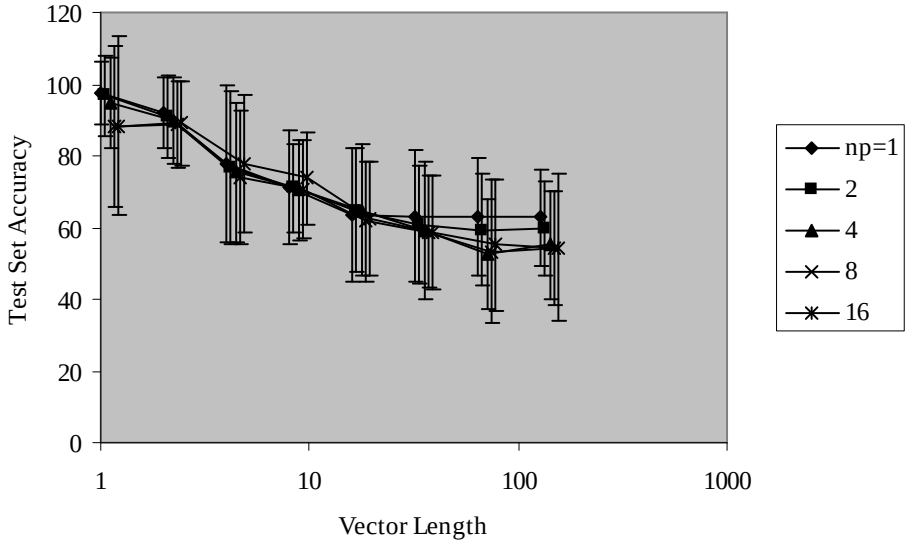

Figure 6.3.9: Parallel AIRS: Accuracy when Varying the Length of the Input Vectors (x-axis offset applied for visual clarity)

Nevertheless, as seen when varying the number of input vectors, there are still gains to be had through the use of this parallel version of AIRS. Interestingly, the number of features has a fairly large impact on AIRS's performance as a classifier as shown in tables D.0.28 and D.0.29 and figures 6.3.9 and 6.3.10 Table D.0.28 shows that as the number of features increases the accuracy of AIRS decreases; however, there is a less obvious trend with respect to the number of memory cells developed (see figure 6.3.10).

Varying the Number of Classes

For this final scalability test, we examined the use of the class-division style of parallelization on data sets with multiple-classes. Table 6.3.1 gives the results on the data sets in [51] which have varying number of classes.

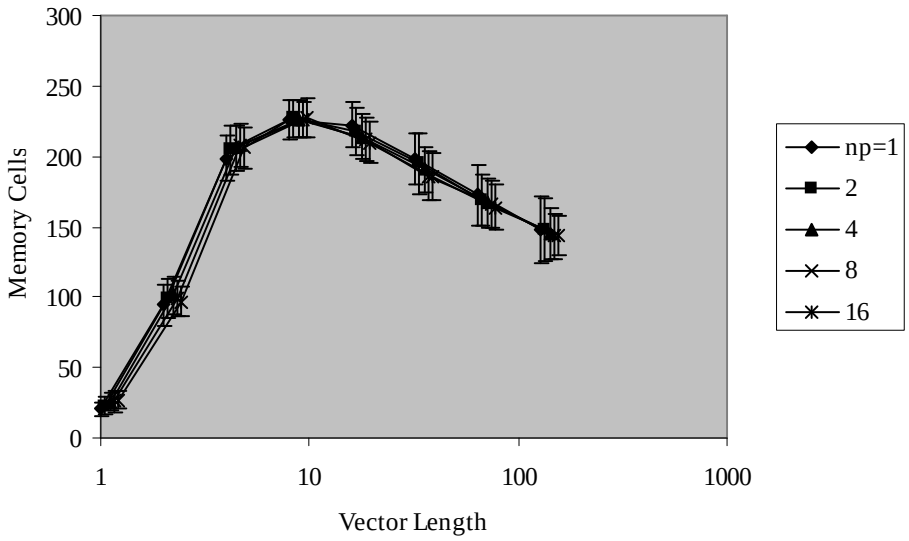

Figure 6.3.10: Parallel AIRS: Memory Cells when Varying the Length of the Input Vectors (x-axis offset applied for visual clarity)

These results indicate that some computational gains can be had for problems with multiple-numbers of classes using our class-division style of parallelization. Of course, these data sets were only two-featured data sets; still, this benefit should only increase with an increase in the feature space. One slightly misleading feature of table 6.3.1, however, is that the data set had an even distribution of the data across all of the classes. In most real-world applications this is seldom the case. Interestingly, we do not find the most efficient use of our processing power when setting the number of processors used equal to the number of classes as we might first suspect. Yet, we do find the most overall decrease in runtime by using the maximum number of processors possible

Table 6.3.1: Parallel AIRS: Class Parallelization

	np	time (s)	S	E_p
3-classes	1	2.96(0.24)	1.00(0.00)	1.00(0.00)
	2	2.41(0.04)	1.23(0.10)	0.61(0.05)
	3	1.78(0.04)	1.66(0.14)	0.55(0.05)
5-classes	1	2.91(0.04)	1.00(0.00)	1.00(0.00)
	2	2.19(0.05)	1.33(0.03)	0.67(0.02)
	4	1.83(0.03)	1.59(0.04)	0.40(0.01)
	5	1.51(0.04)	1.92(0.05)	0.38(0.01)
8-classes	1	2.91(0.03)	1.00(0.00)	1.00(0.00)
	2	2.26(0.03)	1.29(0.02)	0.64(0.01)
	4	1.66(0.05)	1.76(0.05)	0.44(0.01)
	8	1.50(0.05)	1.94(0.06)	0.24(0.01)
12-classes	1	2.92(0.05)	1.00(0.00)	1.00(0.00)
	2	2.11(0.04)	1.38(0.03)	0.69(0.02)
	4	1.62(0.04)	1.81(0.05)	0.45(0.01)
	8	1.46(0.11)	2.00(0.10)	0.25(0.01)
	12	1.27(0.06)	2.29(0.10)	0.19(0.01)

6.3.3 Summary and Discussion of Scalability Experiments

In this section we examined parallel AIRS in terms of its scalability with respect to the number of input vectors, the number of features, and the number of classes. While the results, in general, were not very encouraging with respect to the ability of AIRS to scale to large-scale data sets, there were runtime benefits despite this potential instability. We find decreases in overall runtime by increasing the number of processors. Yet, the efficient use of the parallel resources does not keep pace with the increased overhead from using these resources. This is particularly true of the parallel schemes that require a complex memory cell merging procedure. For the tests examining data sets with multiple classes, we find that the class-division form of parallelization offers more efficient use of the processing power.

The tests sets used here, however, were artificial and designed to distribute the classes evenly across the data set. While it is encouraging that this form of parallelization provides stable gains, most real-world data is not as "neat" as these test sets. In general, as we saw with our verification experiments, our current models of parallelizing AIRS appear too limited to fully take advantage of the additional computation resources. Still, there are gains to be had, and the accuracy has been maintained while achieving this speedup.

6.4 Summary

Our goal with this chapter was to explore ways of exploiting parallelism in an artificial immune system for decreased overall runtime. Using a very basic mechanism for this parallelism, we have seen that there are modest benefits (computationally, at least) from this exploration. One side-effect of our parallelization of AIRS was that its final predictive model increased in size. We explored mechanisms for reducing this size to something more comparable with the serial version.

The need to gather the memory cells into a central location proved to be a bottleneck for our algorithm. The various mechanisms we examined for merging the memory cells developed at distributed sites to our root process did not provide completely satisfying results, but they did offer an initial blueprint for experimenting with this standard computing technology. This raises the question of the necessity of gathering the memory cells at all. The biological immune system

is inherently distributed, and yet, it is still capable of successfully identifying antigens and reacting with them in an appropriate manner. In the next chapter, we explore a basic way of exploring this distributed nature by adapting AIRS to be a more distributed algorithm. This proves to provide greater computational gains than our current parallel algorithm. Additionally, it raises a host of new questions concerning the possibilities of this technique for use as a learning algorithm.

Chapter 7

Distributed AIRS

As we saw in the last chapter our initial approaches to utilizing multiple processors on the AIRS learning algorithm provided mixed results. While the methods explored allowed us to maintain classification accuracy and provided modest computational gains, the parallel efficiency of those modifications were less than desirable. Additionally, three of the methods had to address the question of memory cell merging once the individual processes had finished training. Keeping with our theme of the developmental process of a biologically-inspired algorithm, in this chapter we return to the next source of inspiration from biology: the decentralized nature of the immune system. While we still are exploiting standard computational techniques, we introduce a distributed processing version of AIRS that allows us to explore this biological concept in more detail.

In our previous parallelization schemes we were extremely focused on the gathering and merging of the memory cells at the root process. We felt this memory cell merging was necessary in order to preserve the final predictive model

that was so attractive in the serial version of AIRS. However, if we remove this self-imposed goal of maintaining a single pool of memory cells, we can explore more interesting options.

This chapter examines a different slant on parallelizing the AIRS algorithm. Rather than viewing the goal of our parallelization as developing a memory representation identical to the serial version, we now choose to exploit the behavior of the entire parallel system. If we treat the entire parallel system as a learning model, rather than just a memory cell generation factory, we can offer much more solid computational gains. This approach also has the attractive feature of pointing us to a more distributed AIS algorithm. Since biological immune systems do not have one central antigen identification location, we would like to explore algorithms that also are divorced of this idea. This decentralized view would also allow us to explore more fully the ability of the system to develop a localized response. This would be achieved through localized learning, as well. So beyond just a global evaluation, the local evaluation can provide interesting information about the nature of the learning system, also.

We begin this chapter by providing an overview of this new distributed version of AIRS. We highlight here the changes made to our parallelization of AIRS and the implications these have in terms of classification. We follow this with a series of experiments that echo the ones presented in chapter 6. We find that our computational gains are much greater through this distributed approach and that our new version of AIRS is much more scalable. Yet these changes present several question with regards to the purpose of the AIRS algorithm. We conclude

this chapter with a discussion of how this new version of AIRS provides for the potential for more biologically plausible immune algorithms.

7.1 A Distributed Approach

As previously mentioned, the reaction of the biological immune system to incoming antigens takes place in numerous places throughout the body. Unlike the nervous system which is centrally located, there is no one site that we can point to as the source of all immune responses. Immunological components circulate throughout the system and react in place as needed. We would like to be able to capture this idea in our artificial immune algorithms as well. Our new approach to exploiting increased computational resources for AIRS begins this process.

We start with a simple model of distribution. For this method, we begin as with the previous parallel version of AIRS by distributing a random subset of the training data to each process. We continue, as before, by allowing each process to react to this subset of data with the usual AIRS training routine. However, unlike in our previous approach, we no longer gather the developed memory cells back to the root process. Instead, we now view the entire parallel system as a distributed classifier with multiple classification sites. Some of these sites may be better trained to handle certain classes of the data than others. This is in keeping with our biological systems where there is a need for cell recruitment to certain areas that are under attack if the cells currently at that location are not trained to handle the given invader. We wdo not tackle the issue of recruitment and the

communication of developed memory cells at this time. Rather, we focus only on allowing each processing site to develop its own miniature model of the data set based on its training subset.

For classification, we again distribute the test data throughout the parallel system. This may lead to uneven classification of some data items if the data items are assigned to sites that are not as equipped to recognize them as others sites may be. Nevertheless, we can then evaluate the performance of the system globally on the test data. We can also investigate the local reactions made by each processing site to its assigned pieces of data.

This introduces the need for multiple ways of assessing the performance of the system. As in our previous experiments, one of our chief concerns is the runtime of the system. We want to examine how utilizing more processing power increases our computational gains. We can also still grade the performance of the entire system based on the same classification criterion used earlier. However, the introduction of this distributed memory and reaction model means that we also want to assess the performance at the local level as well. To this extent, we examine the classification capabilities of individual processors on their randomly assigned data. Eventually, this could lead to a network or recruitment based model in which individual populations of memory cells can pass on their classifications and their degrees of confidence in that classification. For now, however, we present a model that is divorced of any interaction other than the initial scattering of the data.

Returning to our MAD framework for learning, we find that the core characterization of AIRS does not change. However, as with our parallel version presented in chapter 6, the final memory model and subsequent decisions based on this model do need to be reevaluated. Each processing site continues to develop its own set of memory cells in the same way as the serial version of AIRS. However, the decision that the system makes concerning a given input is now completely decentralized. While the mechanics of this decision remain the same (i.e., an affinity based approach involving the closest memory cells), the memory structure itself has been altered due to less information available at any one given site. For the current formulation, each site remains limited in this way; however, this is only an initial prototype step. A next step in the evolution of this algorithm could be to incorporate meta-learning strategies or other distributed learning approaches to the disparate memory models [15, 21], or it could be to examine communication strategies available in the immune system, such as cytokine networks or immune network models, to more fully integrate the localized reactions of the system.

7.2 Verification Experiments

We begin by performing similar experiments to those presented in section 6.2 on our three machine learning benchmark data sets. We want to examine both the global and local performance of our new classifier. Additionally, we are still concerned with computational gains, and this is presented here as well.

7.2.1 Experimental Design

For these experiments, we are utilizing the same data sets and processing architecture discussed in section 6.2.1. Again, we perform averages over 30 runs of multiple-way cross-validation.

In order to assess the performance of this new formulation, we offer several metrics. We begin with the global accuracy of the system. This is measured by counting the number of correctly classified test items at each processor and dividing it by the total number of test items distributed to the system. While this global accuracy is not identical to that achieved through a single merged memory cell pool, it does provide a quick overview of how, on average, the system would react to a random data item presented to a random processing site. We then examine the local accuracy. For this, we report the average minimum local accuracy and the average maximum local accuracy. That is, for each run we record which site did the poorest on its assigned test data and which did the best. We also explore the size of the memory model developed. We look globally at the number of memory cells developed throughout the system, and we also look at the minimum and maximum number of memory cells developed at individual sites. Finally, we look at the parallel performance characteristics. In keeping with our goals of achieving faster and more efficient processing, we measure the average run times, speedup, and parallel efficiency of the system.

7.2.2 Results

Tables 7.2.1, 7.2.2, and 7.2.3 give the global accuracy and memory cells developed from this distributed version of AIRS on the Iris, Pima Diabetes, and Sonar data sets.

Table 7.2.1: Distributed Iris: Global Accuracy

np	Accuracy	MCs
1	95.16%(3.06)	63.11(4.70)
2	94.56%(3.98)	74.15(4.53)
4	94.38%(4.59)	84.17(4.12)
8	93.53%(4.04)	95.49(3.56)
16	88.33%(4.88)	104.49(2.91)

Table 7.2.2: Distributed Pima Diabetes: Global Accuracy

np	Accuracy	MCs
1	73.00%(4.40)	279.04(10.11)
2	72.29%(5.00)	317.44(11.00)
4	71.80%(4.75)	358.16(11.39)
8	71.67%(5.06)	400.03(11.63)
16	69.24%(5.15)	445.96(11.06)

Table 7.2.3: Distributed Sonar: Global Accuracy

np	Accuracy	MCs
1	84.79%(8.15)	173.11(3.62)
2	78.81%(9.00)	179.89(3.04)
4	72.61%(11.24)	184.65(2.41)
8	66.97%(11.64)	187.80(1.90)
16	61.94%(12.28)	190.05(1.25)

For all of these results we see a drop-off in this measure of global accuracy as we increase the number of processing sites. However, what this actually

means as far as the classification performance of our new algorithm is less clear. As we mentioned in section 7.2.1, this metric is not exactly equivalent to the accuracy measures for AIRS when using a global memory cell set. This is merely a measurement of the sum of the number of correctly classified items at each processing site divided by the total number of test items distributed throughout the system. Section 7.2.3 provides more discussion into this matter.

Tables 7.2.4, 7.2.5, and 7.2.6 provide the average minimum and maximum test set accuracies and minimum and maximum number of memory cells developed at individual processing sites for distributed AIRS on our three bench mark learning problems.

Table 7.2.4: Distributed Iris: Local Accuracy

np	Min. Acc.	Max. Acc.	Min MCs	Max MCs
1	95.16%(3.06)	95.16%(3.06)	63.11(4.70)	63.11(4.70)
2	90.58%(7.25)	98.53%(2.88)	35.11(2.81)	39.03(2.48)
4	87.36%(8.88)	99.42%(2.65)	18.83(1.43)	23.45(1.52)
8	64.94%(20.24)	100.00%(0.00)	9.56(1.00)	14.03(0.70)
16	19.00%(29.92)	100.00%(0.00)	4.80(0.60)	7.92(0.27)

Table 7.2.5: Distributed Pima Diabetes: Local Accuracy

np	Min. Acc.	Max. Acc.	Min MCs	Max MCs
1	73.00%(4.40)	73.00%(4.40)	279.04(10.11)	279.04(10.11)
2	67.83%(6.20)	76.75%(5.39)	154.71(6.51)	162.73(6.20)
4	60.62%(7.69)	82.40%(5.78)	83.33(3.94)	95.62(3.89)
8	48.87%(9.87)	90.42%(7.05)	43.76(2.68)	55.85(2.39)
16	30.33%(12.71)	99.40%(3.42)	22.73(1.53)	32.98(1.48)

What this metric provides is a glimpse of the range of reactions by individual processors to their assigned data sets. Again, we see the widest range of values

Table 7.2.6: Distributed Sonar: Local Accuracy

np	Min. Acc.	Max. Acc.	Min MCs	Max MCs
1	84.79%(8.15)	84.79%(8.15)	173.11(3.62)	173.11(3.62)
2	70.64%(12.10)	86.99%(9.60)	88.66(2.02)	91.23(1.58)
4	48.40%(18.25)	94.55%(10.64)	44.86(1.04)	47.39(0.63)
8	19.62%(24.44)	100.00%(0.00)	22.39(0.65)	24.00(0.05)
16	0.00%(0.00)	100.00%(0.00)	11.06(0.38)	12.00(0.00)

for the maximum number of processors. We discuss this further in section 7.2.3. Figures 7.2.1, 7.2.2, and 7.2.3 present these accuracy measurements as log-linear graphs with 3σ error bars, and figures 7.2.4, 7.2.5, and 7.2.6 present the memory cells results.

Finally, tables 7.2.7, 7.2.8, and 7.2.9 and figures 7.2.7, 7.2.8, and 7.2.9 provide the parallel performance of the distributed version of AIRS on our three learning problems.

Table 7.2.7: Distributed Iris: Parallel Performance

np	time (s)	S	E_p
1	0.33(0.04)	1.00(0.00)	1.00(0.00)
2	0.17(0.03)	1.96(0.28)	0.98(0.14)
4	0.17(0.07)	2.19(0.77)	0.55(0.19)
8	0.17(0.08)	2.36(1.01)	0.29(0.13)
16	0.22(0.13)	1.84(0.87)	0.12(0.05)

Here we find much more solid parallel gains than seen in chapter 6. Once again, we see inconclusive results Iris data set: it is simply too small and too easy of a classification task to gain much from the use of multiple processors. That is, the time to setup the communication fabric and distribute the data items to the individual processors is greater than the actual time to classify the data. However,

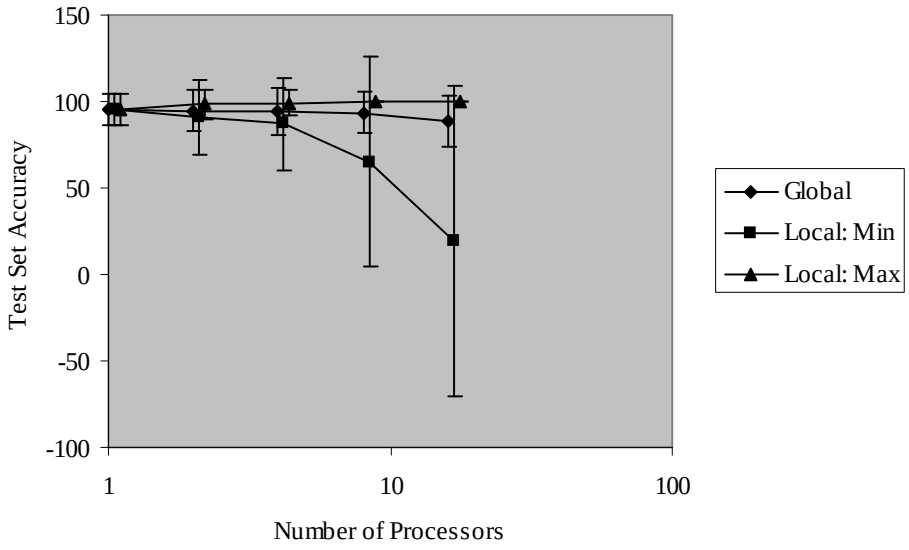

Figure 7.2.1: Distributed AIRS: Iris: Accuracies (x-axis offset applied for visual clarity)

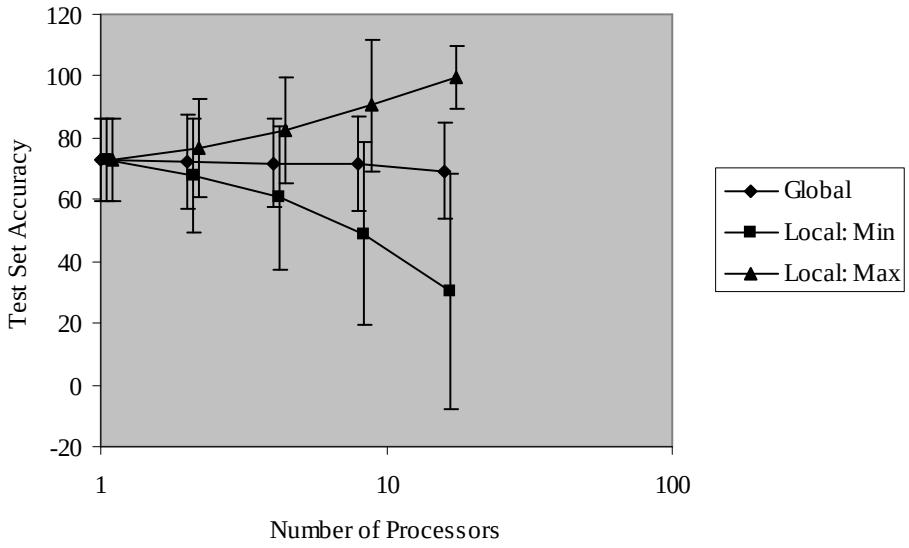

Figure 7.2.2: Distributed AIRS: Pima Diabetes: Accuracies (x-axis offset applied for visual clarity)

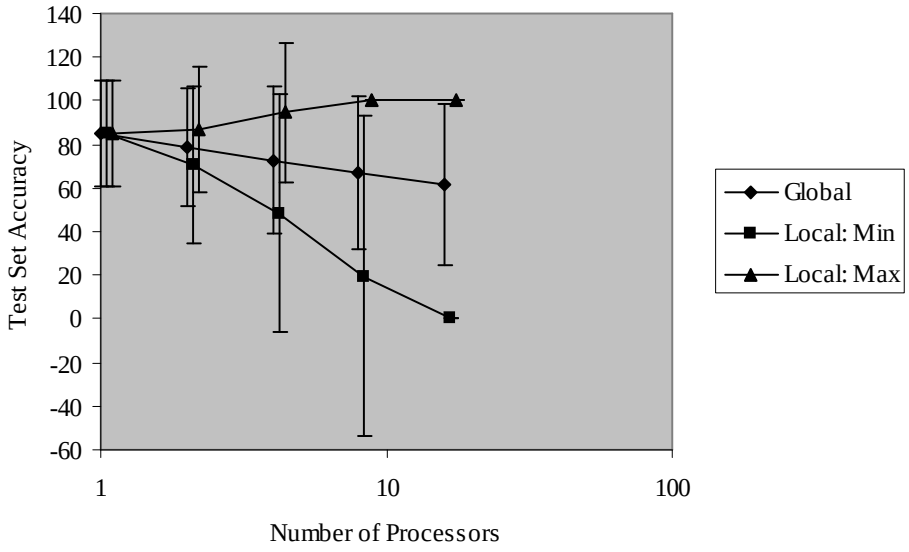

Figure 7.2.3: Distributed AIRS: Sonar: Accuracies (x-axis offset applied for visual clarity)

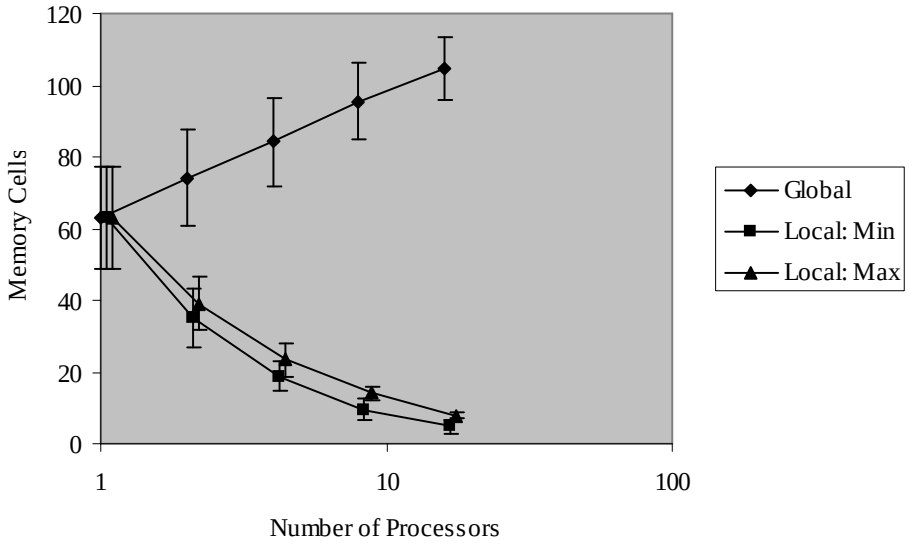

Figure 7.2.4: Distributed AIRS: Iris: Memory Cells (x-axis offset applied for visual clarity)

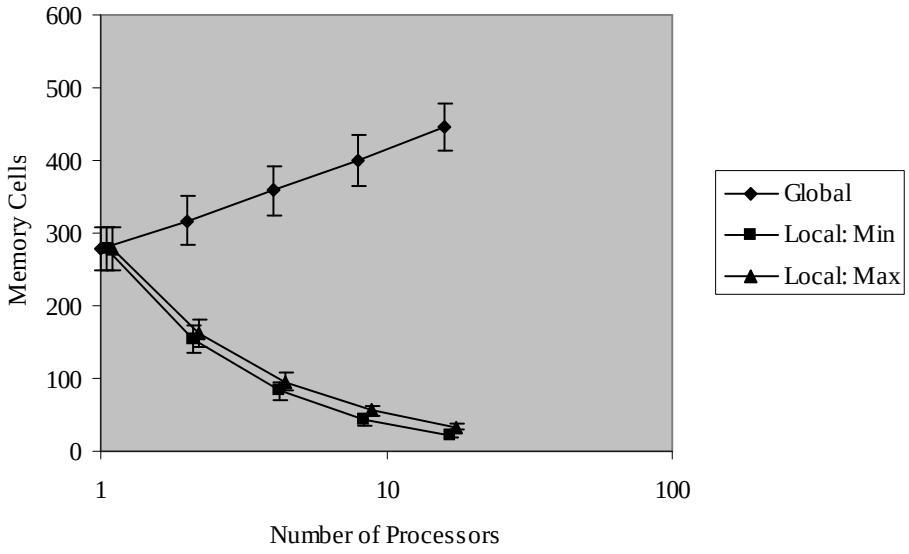

Figure 7.2.5: Distributed AIRS: Pima Diabetes: Memory Cells (x-axis offset applied for visual clarity)

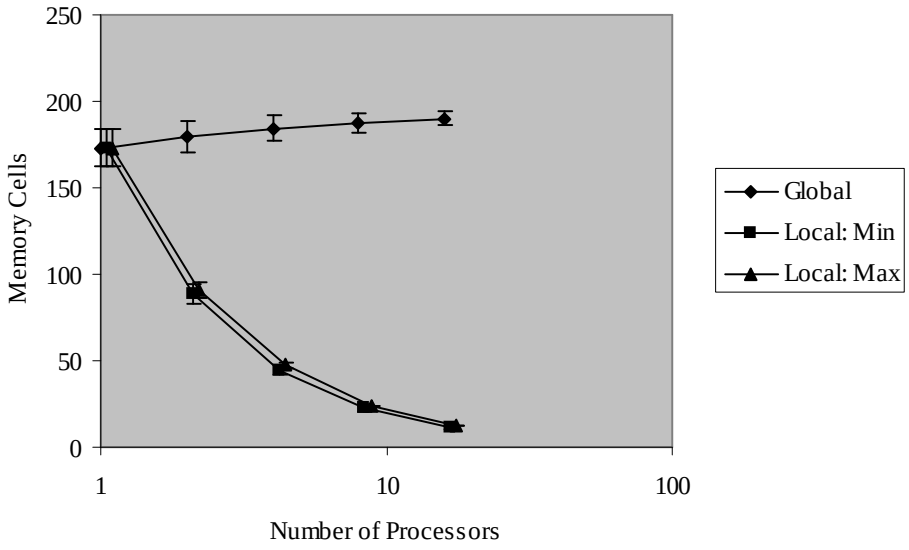

Figure 7.2.6: Distributed AIRS: Sonar: Memory Cells (x-axis offset applied for visual clarity)

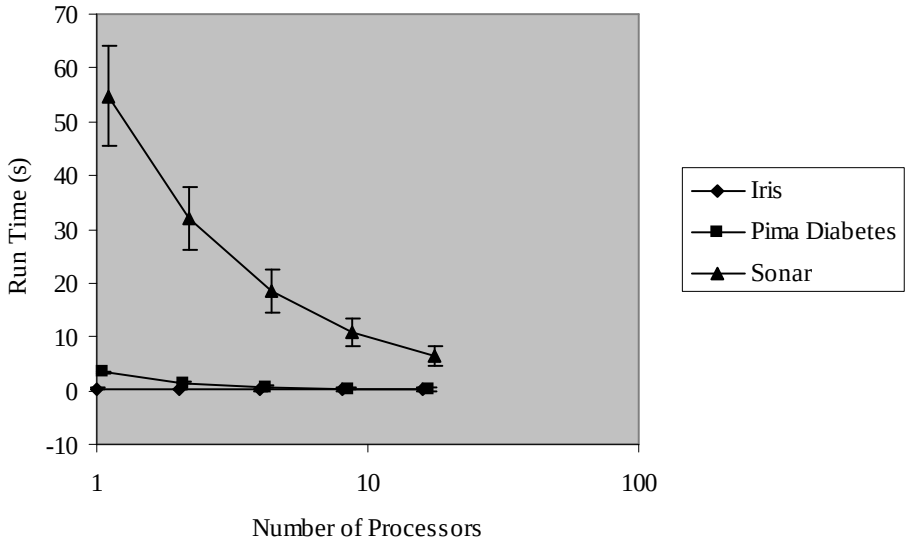

Figure 7.2.7: Distributed AIRS: Run Times (x-axis offset applied for visual clarity)

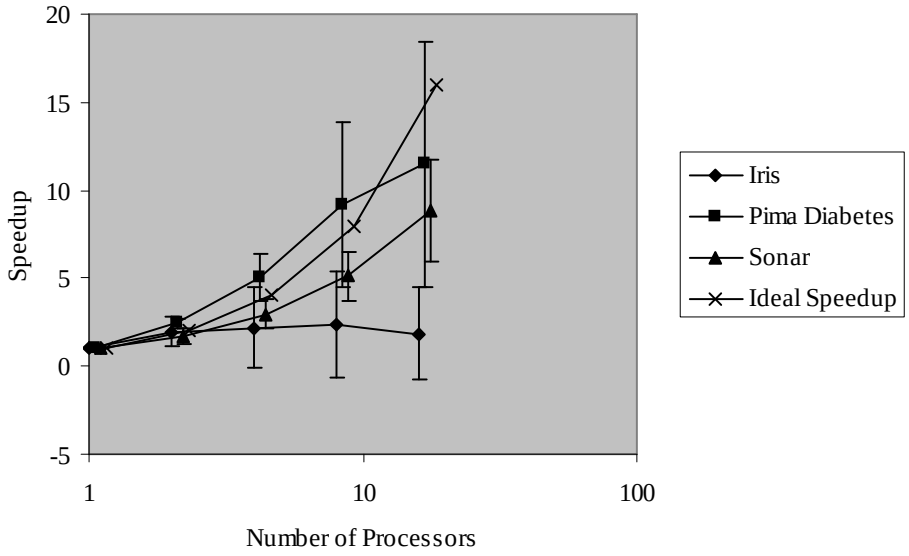

Figure 7.2.8: Distributed AIRS: Speedup (x-axis offset applied for visual clarity)

Table 7.2.8: Distributed Pima Diabetes: Parallel Performance

np	time (s)	S	E_p
1	3.34(0.06)	1.00(0.00)	1.00(0.00)
2	1.36(0.05)	2.46(0.09)	1.23(0.05)
4	0.67(0.07)	5.07(0.44)	1.27(0.11)
8	0.38(0.11)	9.17(1.55)	1.15(0.19)
16	0.31(0.11)	11.46(2.34)	0.72(0.15)

Table 7.2.9: Distributed Sonar: Parallel Performance

np	time (s)	S	E_p
1	54.84(3.07)	1.00(0.00)	1.00(0.00)
2	32.10(1.94)	1.71(0.14)	0.86(0.07)
4	18.50(1.32)	2.98(0.27)	0.74(0.07)
8	10.78(0.81)	5.11(0.47)	0.64(0.06)
16	6.30(0.62)	8.79(0.97)	0.55(0.06)

for our other two data sets we continue to see runtime improvement through the use of our parallelization schemes.

7.2.3 Discussion

The accuracy results presented above raise some potentially troubling questions about this distributed version of AIRS. The behavior exhibited is not surprising. An increase in the number of processing sites decreases the amount of data each site has available for learning. With fewer examples to learn from, the generalizations possible at the individual site are much more limited. That is, with a more incomplete picture of the world, each site's model of the world is also less complete. So, how then, should we view these results? Do they indicate that this approach is useless?

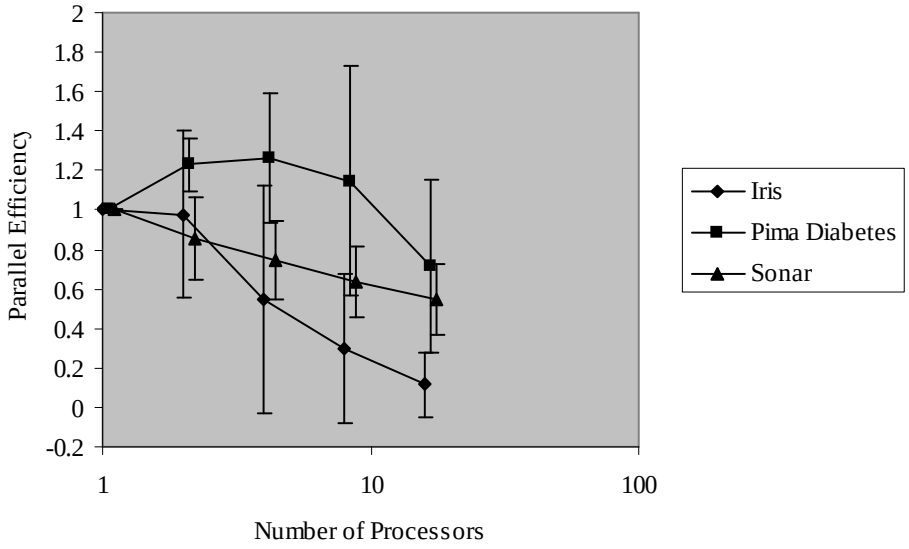

Figure 7.2.9: Distributed AIRS: Parallel Efficiency (x-axis offset applied for visual clarity)

One way of addressing these issues would be through a re-formulation of the approach. Looking at tables E.0.1, E.0.2, and E.0.3 and figures 7.2.10, 7.2.11, and 7.2.10 we find that the distributed version of AIRS is capable of classifying the training data, both at a global and at a local level.

Since AIRS is a supervised learning algorithm, we could embody this knowledge somehow at each processing site. That is, each site can keep track of what type of data (i.e., what classes of data) it has been trained on and its individual performance on that data. This knowledge could then be used in a more global reaction sense. That is, when an individual site is asked to classify some piece of data, it could do so while attaching a degree of confidence to that classification. This degree of confidence could be based on the data's similarity

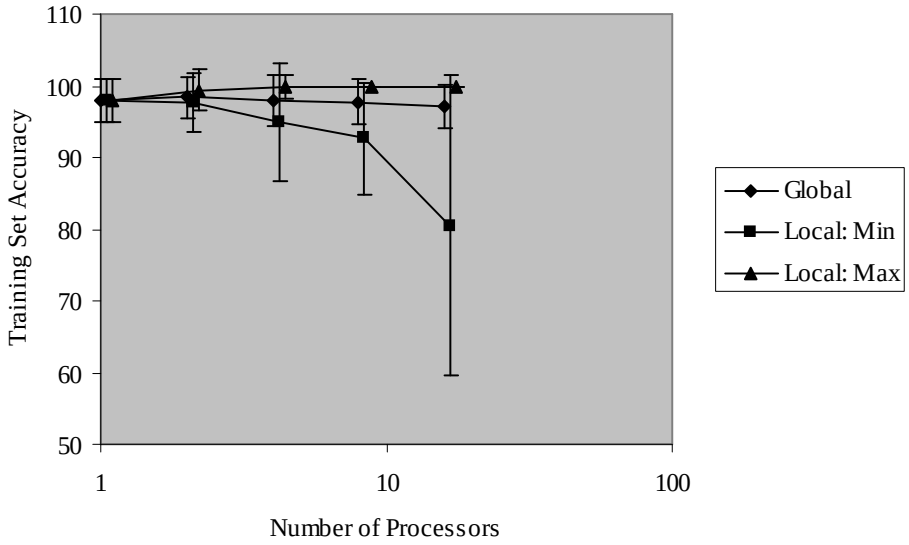

Figure 7.2.10: Distributed AIRS: Iris: Training Set Accuracies (x-axis offset applied for visual clarity)

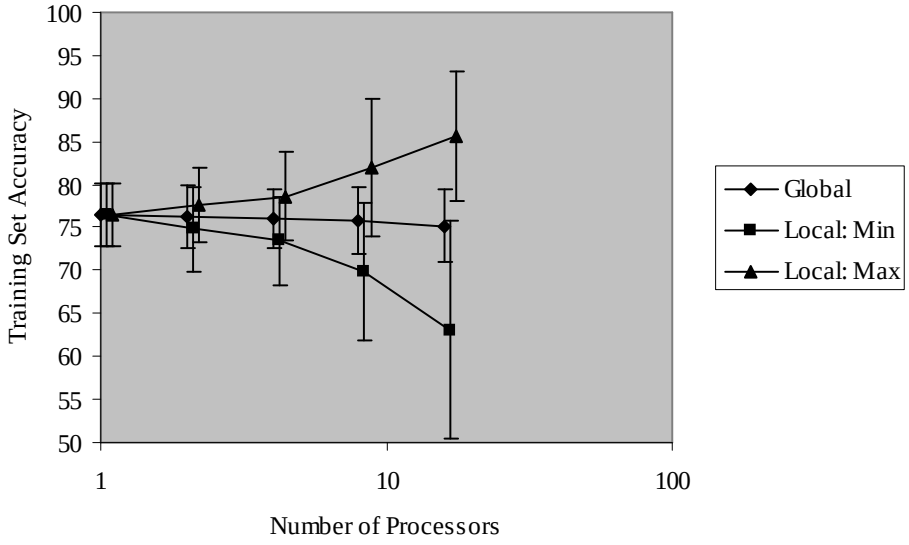

Figure 7.2.11: Distributed AIRS: Pima Diabetes: Training Set Accuracies (x-axis offset applied for visual clarity)

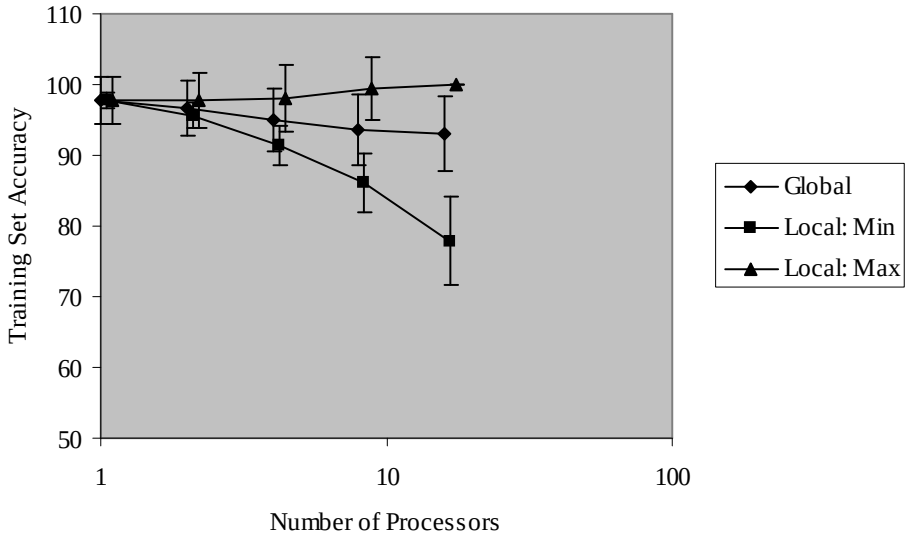

Figure 7.2.12: Distributed AIRS: Sonar: Training Set Accuracies (x-axis offset applied for visual clarity)

to the training data seen at that particular site as well as the individual site's ability to classify that training data. The site could then pass on this confidence to other sites within the system. Basically, what we are proposing is that there could be more interaction among the processing sites, rather than the simple limited isolation in the current model. This approach would not be an attempt to reformulate a global memory cell pool presented in chapter 6. Rather, we could begin to model local interactions, define areas of communication, and introduce concepts of a topology of reaction to a given test data item. This would allow certain sites to share information with other sites, which is more akin to the biological model, while limiting the need for global communication, which is not as biologically plausible.

7.3 Scalability

This section presents scalability tests of distributed AIRS. Again, our chief concern is parallel performance.

7.3.1 Experimental Design

For these tests we used the same data sets as in section 6.3. Again we performed 30 runs of 5-fold crossvalidation. We examined the parallel performance of distributed AIRS as we varied the number of training vectors and the number of features in these training vectors.

7.3.2 Results

Varying the Number of Training Items

Table E.0.4 and figure 7.3.1 give the runtimes and table E.0.5 and figures 7.3.2 and 7.3.3 provide the speedup and parallel efficiency measures on the simulated data for distributed AIRS when varying the number of training instances. The number of test data items is $\frac{N}{4}$. Unlike the similar results presented in section 6.3, these experiments show that the distributed version of AIRS is scalable in terms of the number of training items used. Again we see that an increase in the number of training vectors increases the runtime, but this increase can be counteracted by an increase in the number of processors used. Interestingly, while we see very small runtimes for these experiments, we also find a high degree of variability as noted by the large error bars. In fact, for some of the runtimes there is no

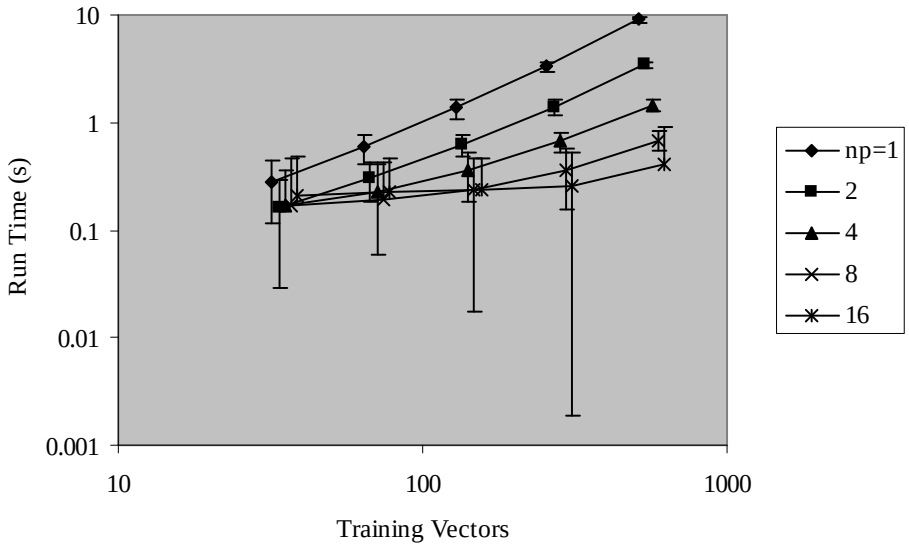

Figure 7.3.1: Distributed AIRS: Run Times when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

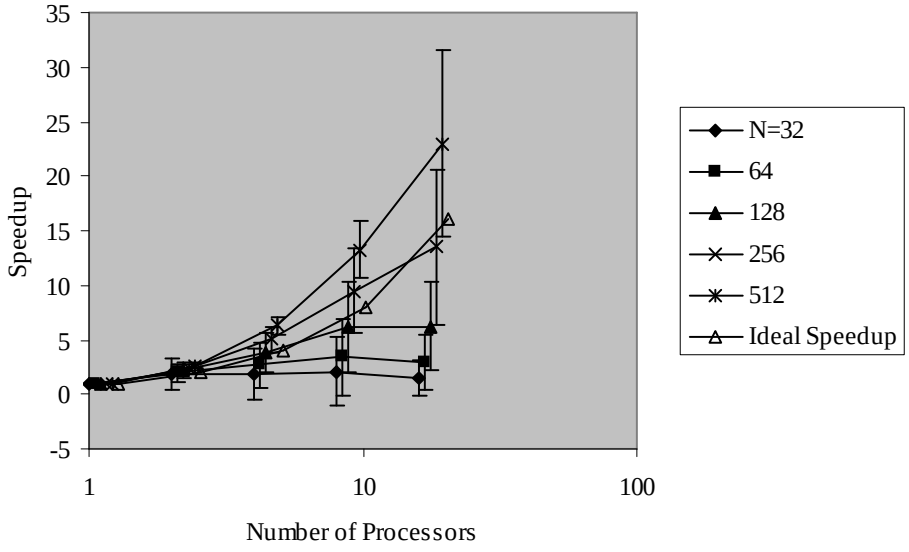

Figure 7.3.2: Distributed AIRS: Speedup when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

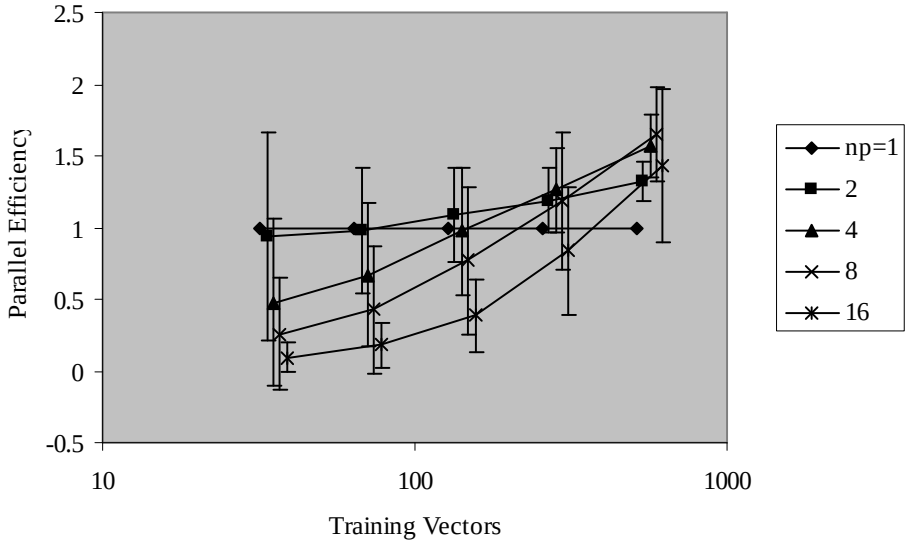

Figure 7.3.3: Distributed AIRS: Parallel Efficiency when Varying the Number of Training Vectors (x-axis offset applied for visual clarity)

distinguishable difference between running the same dataset on more processors. Still, the general trend with these results is that an increase in processing power coupled with an increase in data size leads to scalable efficiency

Varying the Number of Features

Table E.0.6 and figure 7.3.4 give the run times and table E.0.7 and figures 7.3.5 and 7.3.6 provide the speedup and parallel efficiency measures on the simulated data for distributed AIRS when varying the length of the input vector. As with the results seen when varying the number of input items, we find that our distributed version of AIRS appears to scale well when increasing the number of features in the data set. The fundamental relationship between the runtime and the number of features has not changed. That is, with an increase in the number

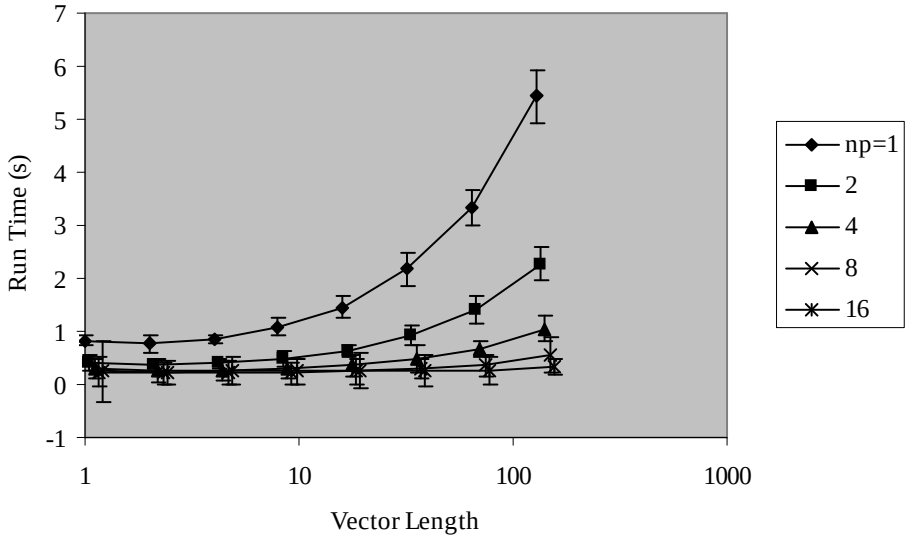

Figure 7.3.4: Distributed AIRS: Run Times when Varying the Length of the Input Vectors (x-axis offset applied for visual clarity)

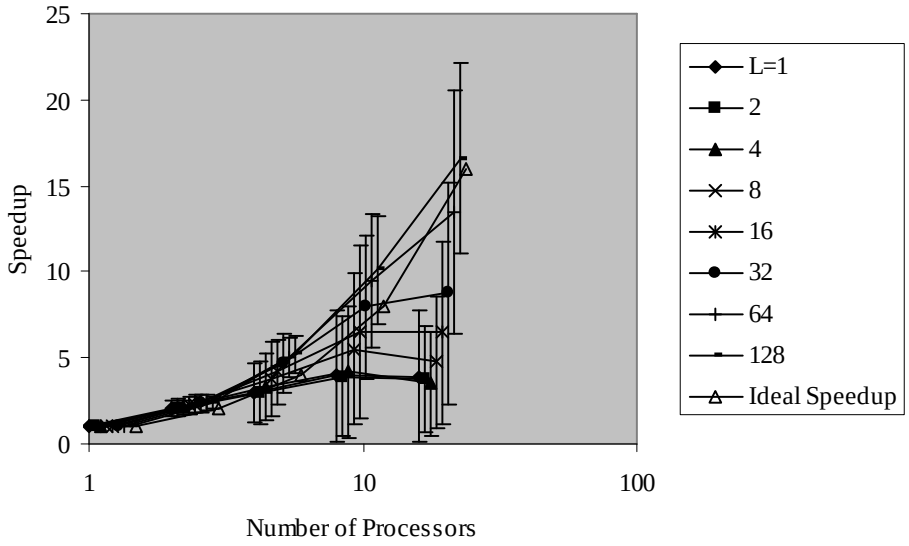

Figure 7.3.5: Distributed AIRS: Speedup when Varying the Length of the Input Vector (x-axis offset applied for visual clarity)

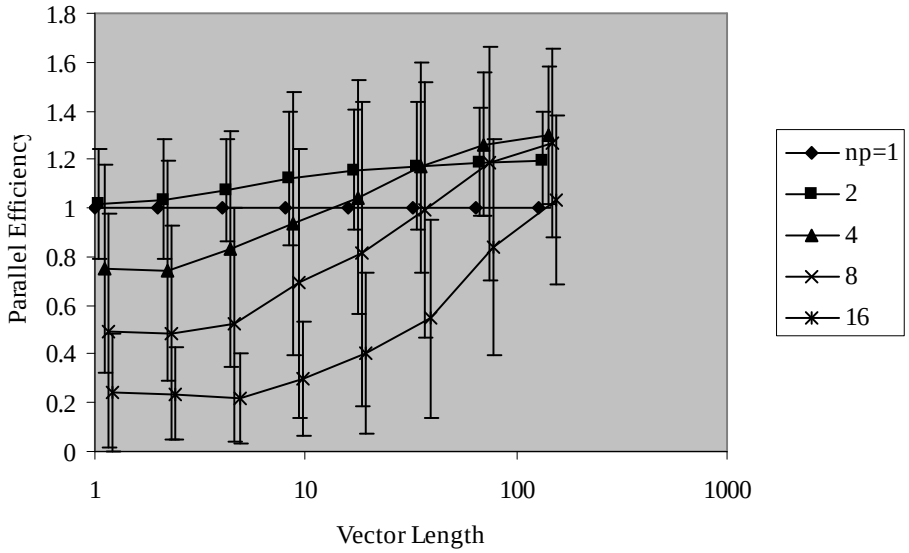

Figure 7.3.6: Distributed AIRS: Parallel Efficiency when Varying the Length of the Input Vector (x-axis offset applied for visual clarity)

of features in the data set there is a corresponding increase in the runtime of the system. However, unlike the results seen for the parallel version presented in section 6.3, these results indicate scalability in the distributed system.

7.3.3 Discussion

This section has demonstrated that our distributed version of AIRS is much more scalable in terms of the number of training items and number of features in the data set than was parallel AIRS. Tables E.0.8 through E.0.19 in appendix E provide the global and local accuracy measures for these experiments along with the number of memory cells developed. As discussed in section 7.2.3, the interpretation of these "accuracy" numbers is somewhat problematic. However,

what they do reveal is this continued sense of a local reaction. This lack of global interaction together with the development of locally learned models and local reactions is much more biologically appealing. Couple this with the stable performance gains, and the distributed version of AIRS offers interesting areas of exploration. Still, one of the key issues that must be addressed is how to utilize these local reactions for solving real-world problems.

7.4 Discussion

This distributed version of AIRS presented in this chapter offers a basic alternative model to using multiple processors when compared to the parallel version discussed in chapter 6. While this model is undeniably faster, its usefulness as a classifier is much more difficult to assess. One of the goals from using this approach was to remove the need for global interaction that was present in previous parallel versions. By doing this, we begin to explore the distributed concepts exhibited in biological immune systems. However, this also possibly limits the predictive capabilities of the system. This distributed design presented here is, admittedly, limited in scope. It removes all interactions among processing sites in the development of individual miniature world views at each site. One way of extending this work and recapturing some of the predictive capabilities of AIRS would be to allow for communication among the sites. While all of the interaction in our parallel model occurred on a global level, we could develop more local interactions. In this way we could begin to simulate cell recruitment and the

diversity of reactions seen in the biological system. While this again may impact our overall runtime, it may offer us more insight into our learning task.

Another key aspect that this distributed approach could allow us to investigate is that of emergence. Much of the field of immune-inspired learning has focused on the engineering of desired behavior into the given system. If what is needed is a classification algorithm, then the biological metaphors are manipulated or engineered to provide this behavior. However, this is counter-intuitive to the development of the biological system itself. Within the immune system the properties that computer scientists find so attractive are in fact emergent properties of the system as a whole. It is through the distributed, diverse reactions of the system that the cognitive capabilities of learning emerge. A distributed approach to learning with local reactions leading to global interaction can provide a truer path to exploring ways that these attractive characteristics evolve and emerge apart from the *a priori* intent engineered into such a system.

7.5 Summary

This chapter presented a distributed version of AIRS as an alternative approach to utilizing multiple processes in an AIS learning algorithm. The results presented here indicate faster processing of the data. However, this decrease in run times is also associated with a seeming decrease in classification accuracy. We have argued that this distributed approach should be seen as a first step along the path to developing more complex and diverse immune algorithms. The biological

immune system is decentralized in its reactions, and we have begun to capture that idea here. The next chapter expands on these areas of future work and offers concluding remarks for this thesis.

Chapter 8

Conclusions and Future Work

We began this discussion by arguing that the development of thinking, intelligent machines has been a fundamental quest for computer science since virtually its inception. One of the hallmarks of human intelligence, we claim, is the ability to learn—that is the ability to improve with experience. The field of machine learning is vast and has been widely studied. Yet, we are always looking for better learning programs. We do this for a couple of reasons. One reason is that we have real world problems that we would like to have computers solve for us. Many of these problems would be too labor-intensive (and possibly monotonous) to have human experts take the time to address them. This is particularly true given the large amounts of digital data that we are now able to collect. Having our machines solve these problems automatically, or at least partially solve them, would be invaluable. The second reason is an inquiry into the nature of learning and intelligence itself. Simply through the process of developing programs that

learn, we discover more about the nature of our own intelligence and capabilities to learn.

We spent the majority of this thesis looking at one avenue for the development of learning algorithms: we examined how the immune system seems to exhibit many characteristics that are necessary for learning. We then showed how these mechanisms and characteristics could be harnessed as metaphors for developing learning algorithms. This chapter provides a summary overview of this investigation into the use of immunological metaphors for developing learning algorithms, details the contributions offered by this thesis, and points to avenues for the future of this work.

8.1 Goals Revisited

The hypothesis of this work was that the mechanisms of the human immune system coupled with standard computing technology could be exploited in the development of machine learning algorithms. While investigating this hypothesis, we identified the need within the field of immune-inspired learning for the exploration of the use of parallel and distributed computing techniques. To achieve our goals, we examined basic immunological components and then answered the question "How can these be used for learning algorithms?" We proposed serial, parallel, and distributed learning algorithms that demonstrated that immunological metaphors can be used to develop successful learning algorithms.

8.2 Summary of the Thesis

8.2.1 Immune Learning

We began this thesis by providing an overview of the biological immune system with the bias of identifying those components that exhibit learning. Since learning is an adaptive process, we chose to focus on the adaptive layer of the immune system. Within this layer we detailed the workings of the B- and T-cell lymphocytes. As with other immune-inspired learning algorithms, we narrowed the scope of our discussion primarily to the workings of the B-cells. B-cells develop through a process of clonal selection and somatic hypermutation. These two processes were proposed to be key mechanisms for learning. However, learning is more than just adaptation. In order to provide rapid secondary responses to previously seen antigens, the adaptive immune system develops a set of memory cells which are more easily stimulated upon encountering an antigen. We identified this memory mechanism as key to the immune system's ability to improve with experience.

This led to the identification that any learning system is composed of three components: memory, adaptation, and decision-making. We used these three components as a simple framework for discussing learning. In doing so, we were able to discuss a wide-variety of learning algorithms—both biologically-inspired and more traditional.

This was followed by a review of immune-inspired learning algorithms. Several avenues of investigation have been pursued for using immunological metaphors

for learning. These were divided into population-based and network-based algorithms. Within the population-based algorithms, several different immune mechanisms have been explored: gene libraries, negative selection, clonal selection, and memory cell development. The network-based algorithms have focused on the use of Jerne's immune network theory to model the interactions of the immune system. Immunological memory is maintained within the network of connections among the immune cells. We recognized the fact that very little had been done with immune-inspired **supervised** learning. This led to the development of the AIRS learning algorithm which we studied in-depth.

8.2.2 AIRS: A Immune-Inspired Supervised Learning Algorithm

The Artificial Immune Recognition System (AIRS) immune learning algorithm developed from the identification that immunological metaphors had not been exploited for the field of supervised learning. The initial work on this algorithm demonstrated that exploiting such immunological metaphors as clonal selection, B-cell growth and death, affinity maturation, and memory cell development could lead to the development of a successful supervised learning algorithm. However, the initial algorithm contained some unnecessary complications. In this thesis, we identified those complicating factors and proposed ways to simplify the algorithm. In doing so, we were also able to introduce another immunological component that the original algorithm was lacking: affinity-based hypermutation.

We demonstrated that, by using the affinity of a cell to guide its mutation, a more efficient memory model was developed. With this change, we found that the quality of the classification algorithm remained the same while its data-reduction capabilities increased. With AIRS we demonstrated how immunological metaphors can be used in the development of a serial learning algorithm.

8.2.3 Parallel and Distributed Learning

We spent the last half of this thesis focused on the use of more processing power. Since the immune system is distributed throughout the body with no central processing site, we wanted to explore how we could use multiple processors for the development of immune-inspired learning algorithms. Initially, however, our primary focus has been computational gains rather than behavioral changes.

We began with the simple CLONALG immune-learning algorithm. This population-based algorithm focuses on the development of memory cells with no real interaction among the immunological components. This lack of interaction made it an ideal candidate for our initial parallel investigations. We demonstrated that, by scattering the training data to various processors and allowing each processor to develop its separate memory model of the given data speedup, in the overall processing time is achieved. We showed that this basic parallelization is also scalable in terms of the number of input items and the number of features in the input space.

We then took this basic method of parallelization and applied it to the AIRS algorithm. One major issue with the AIRS algorithm, however, is that there is a degree of interaction among the cells. This led to the need to devise a means of recapturing the predictive model of AIRS after the development of memory cells had occurred at individual processing sites. Several methods were proposed, and while none of them were wholly satisfactory, we did find that using multiple processors in this manner provided a decrease in runtime for many data sets.

We concluded our investigations of using multiple processors for the AIRS algorithm by proposing a simple distributed approach. As it is formulated, this distributed approach is of questionable use. However, this simple model is presented as the first step in a line of investigation of ways to remove the need for central control within this immune-inspired learning algorithm. In section 8.4, we discuss how this line of work might continue.

8.3 Contributions

This thesis has provided the following contributions to the fields of immune-inspired computing and machine learning:

- The components of the immune system that exhibit the capacity for learning were detailed.

- A framework for discussing learning algorithms was proposed. Three properties of every learning algorithm—memory, adaptation, and decision-making—were identified for this framework, and traditional learning algorithms were placed in the context of this framework.

- An investigation into the use of immunological components for learning was provided. This led to an understanding of these components in terms of the learning framework

- A simplification of the AIRS immune-inspired learning algorithm was provided by employing affinity-dependent somatic hypermutation.

- A parallel version of the CLONALG immune learning algorithm was developed. It was shown that basic parallel computing techniques can provide computational benefits for this algorithm.

- A parallel version of AIRS was offered. It was shown that applying these same parallel computing techniques to AIRS, while less scalable than when applied to CLONALG, still provided computational gains.

- A distributed approach to AIRS was offered, and it was argued that this approach provided a more biologically appealing model. The simple distributed approach was proposed in terms of an initial step toward a more complex, distributed system.

8.4 Future Work

This thesis has investigated ways of using the immune system as a source of inspiration for the development of learning algorithms. In doing so, it has focused on fairly specific examples of accomplishing this. It has hinted at a general means for using parallel processing techniques in the development of more efficient learning algorithms as well. All of this can be seen as proof-of-concept type activities. As such, there is inevitably more to be done. We discuss some of these ideas here.

8.4.1 Theory of Convergence

In our asymptotic analysis of the AIRS algorithm presented in section 3.5.3, we were unable to completely simplify the runtime analysis in terms of the input size. This was due to no general way of predicting the number of training generations needed before converging on an appropriate candidate memory cell. We again saw this issue when analyzing the CLONALG algorithm. This points to the need for theoretical models of convergence for immune-inspired learning algorithms. Although the authors of [123] offer one model based on Markov chains for the convergence analysis of a multiobjective immune-inspired algorithm, little else has been done in this area. While some ideas may be borrowed from convergence studies in genetic algorithms, there is enough difference in the behavior of immune-inspired algorithms to warrant further investigation.

8.4.2 Diversity of Cellular Types

As we saw in our overview of immune learning algorithms, the majority of such algorithms have focused only on one type of immune cell. Yet, the immune system contains a range of diverse cell types. There seems to be a need to look beyond just single- cell models of inspiration. As we branch further out and look to developing more complex, multi-cellular immune learning algorithms, we might discover more interesting behavioral properties than our limited approaches currently provide. Since each cell type performs a different function that is independent of a global control yet leads to a cooperative reaction to a given stimulus, we could, again, explore the ideas of emergent and cooperative behavior by exploiting more diverse cellular metaphors in our artificial systems.

These ideas are echoed in [108] when the authors identify five key areas that affect the behavior of complex systems. These areas include "openness, diversity, interaction, structure, and scale." With an increase in cellular types modeled along with the ability to simulate localized response and learning, we can begin to explore the impact these areas have on the development of our biologically-inspired systems.

8.4.3 Distributed Learning

As we saw in chapter 7, taking a purely distributed approach to our learning algorithms presents a change in the behavior of the algorithm. There is a wealth of

possibilities to explore in this area. Some of this has been hinted at in the immune-inspired security work where anomaly detection sites are scattered throughout a protected network [72, 61]. However, we can take this further.

By focusing on the distributed nature of the immune system, we can model such concepts as cellular recruitment and local reactions. One possibility would be for clusters of computing nodes to become "experts" on certain types of data. Utilizing the supervised learning ideas from AIRS, a group of nodes could learn what areas of expertise it has. Then, when a new situation arises—a new piece of data is presented to the system—different centers of expertise could contribute its local knowledge to the reaction of the entire system. This also might allow for the integration of multiple domains of knowledge throughout a distributed system or for certain areas to become experts at identifying one class of knowledge within a single domain. Figure 8.4.1 presents a stylized architectural view of this when using our three benchmark domains.

Of course, if we integrate different domains within the system, knowledge representation will become of key importance. Also, this view of distributed learning is much different than the parallel approaches presented in most of this thesis. Our concern would no longer be primarily with increased computational gains, but would be more with diversity of response.

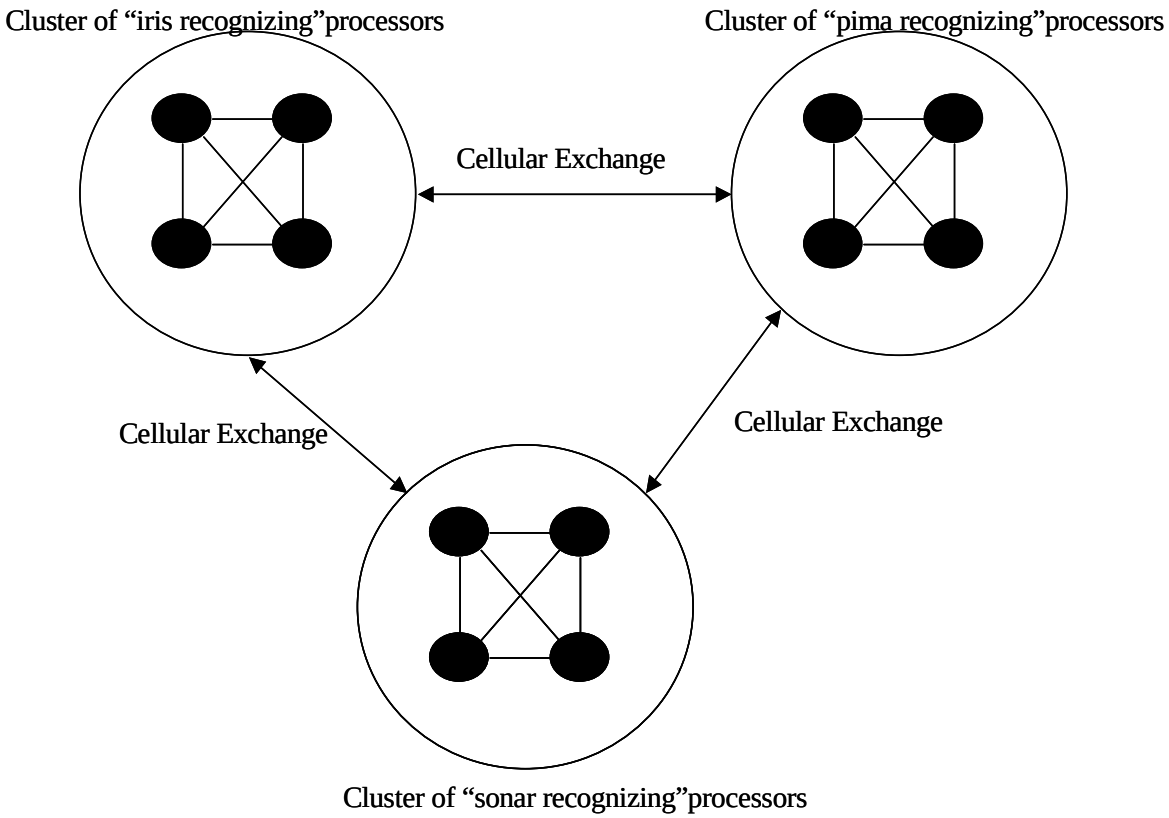

Figure 8.4.1: Possible Architecture for a Multi-Domain Distributed Learning System

8.4.4 Emergence

This diversity of response could also lead to an investigation into emergent properties of such artificial immune systems. As hinted in section 7.4, rather than being pre-engineered, many of the attractive characteristics of the immune system emerge through the interactions of the diverse components. However, little has been done by way of exploration of the mechanisms for emergence. By building truly distributed systems, we can begin to address the questions of how

such interconnected yet distinct reactions can lead to the emergence of global reactions.

8.4.5 Interdisciplinary Research

As we have pointed out in the development of our immune-inspired learning algorithms, the development process often swings back and forth from the biology to standard computing techniques. The potential benefits of the process is limited by the knowledge of the investigators. There is a definite need, from a computer scientist's perspective, for greater interaction with those working in other disciplines. By working with biologists, we can delve deeper into our understanding of the computational processes of the natural systems. This, in turn, can inform and enhance our artificial systems. And by exploring more computational models and techniques in our biologically-inspired systems, we can potentially offer insights to biologists concerning the natural systems. There is the need for the give and take offered by interdisciplinary research as we attempt to explore these non-standard models of computation more fully.

8.5 Concluding Remarks

In this thesis we have shown that the immune system can be used for the development of learning algorithms. We have followed the path of the development of a particular biologically-inspired algorithm and demonstrated how this process moves from biological inspiration to the incorporation of standard computing

technologies and back again. We have seen ways of modifying existing immune learning algorithms to take advantage of multiple processes. And we have examined the beginnings of the development of a distributed immune-inspired learning system. However, is there a point in developing more algorithms based on the immune system? Beyond the proof-of-concept type of activities that have been done to date, why should we continue this line of immune-inspired computational systems building? One way to answer this is by pointing out that the fact that we can build immune-inspired learning algorithms at all is quite fascinating and points to some interesting questions. Is the immune system really a cognitive system? What are the interactions between the immune system and our traditional cognitive system—the nervous system? We argue that by building immune-inspired learning systems we begin to tackle some of these larger biological questions as well. That is, through our system development work, we discover more about what the ingredients of cognition are and where the boundaries of human cognition exist. Also, on a more pragmatic side, the development of immune-inspired machine learning systems might point to more efficient or robust methods of addressing learning problems. We already see hints of this with the algorithms that have been developed to date. AIRS, for example, performs as well as many other machine learning techniques and offers several advantages (such as one-pass learning and large-scale data reduction) that are not present in these other techniques. The immune system seems to have numerous avenues of exploration yet to be tapped for the development of computational systems. As with any field of research, there is no guarantee that this will lead to anything

useful in the long run. Yet, the potential definitely seems to exist, and just the process of building these systems, as with most scientific pursuits, will inevitably point toward answers to some fundamental questions concerning human nature itself.

Bibliography

[1] *Proceedings of the IEEE computer society symposium on research in security and privacy held in Oakland, California, May 6-8, 1996.* Los Alamitos, CA: Los Alamitos, CA, May 1996.

[2] *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ser. Lecture Notes in Computer Science, no. 2723. Chicago: Springer-Verlag, July 2003.

[3] H. Aisu and H. Mizutani, "Immunity-based learning—integration of distributed search and constraint relaxation," in *International Workshop on the Immunity-Based Systems (IMBS 96) part of International Conference on Multi-Agent Systems (ICMAS'96)*, 1996.

[4] D. Allen, A. Cumano, R. Dildrop, C. Kocks, K. Rajewsky, N. Rajewsky, J. Roes, F. Sablitzky, and M. Siekevitz, "Timing, genetic requirements and functional consequences of somatic hypermutation during B-cell development," *Immunological Reviews*, no. 96, pp. 5–22, 1987.

[5] G. M. Amdahl, "Validity of the single-processor approach to achieving large scale computing capabilities," in *Proceedings of the AFIPS Conference*. Reston, VA: AFIPS Press, April 1967, pp. 483–485.

[6] J. M. Baldwin, "A new factor in evolution," *American Naturalist*, vol. 30, pp. 441–451, 1896.

[7] P. Benedict, "Data generation program/2 v1.0," `http://ftp.ics.uci.edu/pub/machine-learning-databases/dgp-2/`, 1990, Inductive Learning Group, Beckman Institute for Advanced Technology and Sciences, University of Illinois at Urbana.

[8] H. Bersini, "The endogenous double plasticity of the immune network and the inspiration to be drawn for engineering artifacts," pp. 22–44, in [25].

[9] H. Bersini, "Self-assertion versus self-recognition: A tribute to Francisco Varela," pp. 107–112, in [115].

[10] H. Bersini and F. Varela, "Hints for adaptive problem solving gleaned from immune networks," in *Parallel Problem Solving from Nature, 1st Workshop*

PPSN 1. Dortmund and Federal Republic of Germany: Springer-Verlag, 1990, pp. 343–354.

[11] H. Bersini and F. Varela, *Computing with Biological Metaphors.* Chapman Hall, 1994, ch. The immune learning mechanisms : Reinforcement and recruitment and their applications, pp. 166–192.

[12] C. M. Bishop, *Neural Networks for Pattern Recognition.* Oxford, UK: Oxford University Press, 1995.

[13] C. L. Blake and C. J. Merz, *UCI Repository of Machine Learning Databases*, 1998. URL: http://www.ics.uci.edu/ mlearn/MLRepository.html

[14] E. Bonabeau, M. Dorigo, and G. Theraulaz, *Swarm Intelligence: From Natural to Artificial Systems.* MIT Press, 2000.

[15] P. Brazdil, M. Gams, S. Sian, L. Torgo, and W. van de Velde, "Learning in distributed systems and multi-agent environments," in *Lecture Notes in Artificial Intelligence Vol.482: Machine Learning-EWSL-91*, Y. Kodratoff, Ed., 1991, pp. 412–423. URL: citeseer.ist.psu.edu/brazdil91learning.html

[16] F. Burnet, *The clonal selection theory of acquired immunity.* Cambridge University Press, 1959.

[17] E. Cantú-Paz, "A survey of parallel genetic algorithms," *Calculators Parallels*, vol. 10, no. 2, pp. 141–171, 1998.

[18] E. Cantú-Paz, *Efficient and Accurate Parallel Genetic Algorithms.* Kluwer Academic Publishers, 2000.

[19] J. H. Carter, "The immune system as a model for pattern recognition and classification," *Journal of the American Medical Informatics Association*, vol. 7, no. 1, pp. 28–41, jan/feb 2000.

[20] P. K. Chan and S. J. Stolfo, "Toward scalable and parallel inductive learning: A case study in splice junction prediction," in *ML94 Workshop on Machine Learning and Molecular Biology*, 1994.

[21] P. K. Chan and S. J. Stolfo, "Toward parallel and distributed learning by meta-learning," in *Working Notes AAAI Work. Knowledge Discovery in Databases*, 1993, pp. 227–240. URL: citeseer.ist.psu.edu/chan93toward.html

[22] J. Chattratichat, J. Darlington, M. Ghanem, Y. Guo, H. Hunning, M. Kohler, J. Sutiwaraphun, H. Wing To, and D. Yang, "Large scale data mining: Challenges and responses," in *KDD-97*, 1997, pp. 143–146.

[23] D. Dasgupta and S. Forrest, "Novelty detection in time series data using ideas from immunology," in *Proceedings of the 5th International Conference on Intelligent Systems.* Reno, USA: AAAI Press, 1996, pp. 87–92.

[24] D. Dasgupta, S. Yu, and S. Majumdar, "MILA—multilevel immune learning algorithm," pp. 183–194, in [2].

[25] D. Dasgupta, Ed., *Artificial Immune Systems and Their Applications*. Berlin: Springer, 1998.

[26] D. Dasgupta and S. Forrest, "An anomaly detection algorithm inspired by the immune system," pp. 262–277, in [25].

[27] L. N. de Castro and J. Timmis, "An artificial immune network for multimodal optimisation," pp. 699–704, in [67].

[28] L. N. de Castro and J. Timmis, *Artificial immune systems: A new computational approach*. London. UK.: Springer-Verlag, September 2002. URL: http://www.cs.kent.ac.uk/aisbook/

[29] L. N. de Castro and F. von Zuben, "The clonal selection algorithm with engineering applications," in *Proceedings of Genetic and Evolutionary Computation*. Las Vegas, USA.: Morgan-Kaufman, 2000, pp. 36–37.

[30] L. N. de Castro and F. von Zuben, "An evolutionary immune network for data clustering," in *SBRN*, Brazil, 2000, pp. 187–204.

[31] L. N. de Castro and F. J. Von Zuben, "aiNet: An artificial immune network for data analysis," in *Data Mining: A Heuristic Approach*, H. A. Abbass, R. A. Sarker, and C. S. Newton, Eds. Idea Group, 2001, ch. 12, pp. 231–259.

[32] L. N. de Castro and F. von Zuben, "Learning and optimization using the clonal selection principle," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 3, pp. 239–251, June 2002.

[33] P. D'haeseleer, S. Forrest, and P. Helman, "An immunological approach to change detection: Algorithms analysis and implications," pp. 110–119, in [1].

[34] W. Duch, "Datasets used for classification: Comparison of results," http://www.phys.uni.torun.pl/kmk/projects/datasets.html, November 2000. URL: http://www.phys.uni.torun.pl/kmk/projects/datasets.html

[35] W. Duch, "Logical rules extracted from data," http://www.phys.uni.torun.pl/kmk/projects/rules.html, November 2000. URL: http://www.phys.uni.torun.pl/kmk/projects/rules.html

[36] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. John Wiley and Sons, 1973.

[37] R. Duncan, "A survey of parallel computer architectures," *IEEE Computer*, pp. 5–16, February 1990.

[38] T. Elomaa and M. Kääriäinen, "An analysis of reduced error pruning," *Journal of Artificial Intelligence Research*, vol. 15, pp. 163–187, 2001.

[39] J. Farmer, N. Packard, and A. Perelson, "The immune system and adaptation and machine learning," *Physica D*, vol. 22, pp. 187–204, 1986.

[40] J. D. Farmer, "A Rosetta stone for connectionism," *Physica D*, vol. 42, pp. 153–187, 1990.

[41] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computing*, vol. C-21, no. 9, pp. 948–960, September 1972.

[42] S. Forrest and S. A. Hofmeyr, "Immunology as information processing," in *Design Principles for the Immune System and Other Distributed Autonomous Systems*, ser. Santa Fe Institute Studies in the Sciences of Complexity, L. A. Segel and I. Cohen, Eds. New York: Oxford University Press, 2001, pp. 361–387.

[43] S. Forrest, S. A. Hofmeyer, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," pp. 120–128, in [1].

[44] S. Forrest, S. A. Hofmeyr, and A. Somayaji, "Computer immunology," *Communications of the ACM*, vol. 40, no. 10, pp. 88–96, October 1997.

[45] S. Forrest, A. S. Perelson, L. Allen, and R. Cherukuri, "Self-nonself discrimination in a computer," in *Proceedings of the IEEE computer society symposium on research in security and privacy held in Oakland, California, May 16-18, 1994*. Los Alamitos, CA: IEEE Computer Society Press, 1994, pp. 202–212.

[46] A. Freitas and J. Timmis, "Revisiting the foundations of artificial immune systems: A problem oriented perspective," pp. 229–241, in [114].

[47] G. Galal, D. J. Cook, and L. B. Holder, "Improving scalability in a scientific discovery system by exploiting parallelism," in *Proceedings of The 1997 Conference on Knowledge Discovery in Databases (KDD97)*, 1997, pp. 171–174.

[48] F. Gonzalez, D. Dasgupta, and L. F. Niño, "A randomized real-valued negative selection algorithm," pp. 261–272, in [114].

[49] D. Goodman and L. Boggess, "The role of hypothesis filter in AIRS, an artificial immune classifier," in *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems and Artificial Life*, C. Dagli, A. Buczak, J. Ghosh, M. Embrechts, and O. Ersoy, Eds. New York: ASME Press, November 2003, vol. 13, pp. 243–248.

[50] D. Goodman, L. Boggess, and A. Watkins, "An investigation into the source of power for AIRS, an artificial immune classification system," in *Proceedings of the International Joint Conference on Neural Networks 2003*. Portland, OR, USA: The International Neural Network Society and the IEEE Neural Networks Society, 2003, pp. 1678–1683.

[51] D. Goodman, L. Boggess, and A. Watkins, "Artificial immune system classification of multiple-class problems," in *Intelligent Engineering Systems Through Artificial Neural Networks: Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Data Mining, and Complex Systems*, C. H. Dagli, A. L. Buczak, J. Ghosh, M. J. Embrechts, O. Ersoy, and S. W. Kercel, Eds. New York: ASME Press, 2002, vol. 12, pp. 179–184.

[52] A. Grama, A. Gupta, and V. Kumar, "Isoefficiency: Measuring the scalability of parallel algorithms and architectures," *IEEE Parallel and Distributed Technology*, vol. 1, no. 3, pp. 12–21, August 1993.

[53] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, 2nd ed. Addison-Wesley, 2003.

[54] J. Greensmith and S. Cayzer, "An artificial immune system approach to semantic document classification," pp. 136–146, in [114].

[55] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message Passing Interface*, 2nd ed. MIT Press, 1999.

[56] J. Hamaker and L. Boggess, "Non-Euclidean distance measures in AIRS, an artificial immune classification system," in *Proceedings of the 2004 Congress on Evolutionary Computation*, June 2004, pp. 1067–1073.

[57] E. Hart and P. Ross, "Studies on the implications of shape-space models for idiotypic networks," pp. 413–426, in [93].

[58] S. Haykin, *Neural Networks: A comprehensive foundation*. Upper Saddle River, NJ, USA: Prentice Hall, 1999.

[59] R. Hightower, S. Forrest, and A. Perelson, "The baldwin effect in the immune system: Learning by somatic hypermutation," in *Adaptive Individuals in Evolving Populations*, R. K. Belew and M. Mitchell, Eds. Addison-Wesley, 1996, pp. 159–167.

[60] R. Hightower, S. Forrest, and A. S. Perelson, "The evolution of emergent organization in immune system gene libraries," in *Proceeding of the Sixth International Conference on Genetic Algorithms*, L. J. Eshelman, Ed. San Francisco, CA: Morgan Kaufmann, 1995, pp. 344–350.

[61] S. Hofmeyr and S. Forrest, "Architecture for an artificial immune system," *Evolutionary Computation*, vol. 7(1), pp. 45–68, 2000.

[62] S. A. Hofmeyr and S. Forrest, "Immunity by design: An artificial immune system," in *Proceedings of the genetic and evolutionary computation conference (GECCO) held in Orlando, Florida, July 13-17, 1999*. San Francisco, CA: Morgan-Kaufmann, 1999, pp. 1289–1296. URL: http://www.cs.unm.edu/

[63] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," http://www.cs.unm.edu/immsec/publications/ids.ps, 1998. URL: http://www.cs.unm.edu/immsec/publications/ids.ps

[64] P. G. Holt and C. A. Jones, "The development of the immune system during pregnancy and early life," *Allergy*, vol. 55, pp. 688–697, 2000.

[65] J. Horn and D. E. Goldberg, "A timing analysis of convergence to fitness sharing equilibrium," *Lecture Notes in Computer Science*, vol. 1498, pp. 23–33, 1998. URL: citeseer.ist.psu.edu/30390.html

[66] J. Hunt and A. Fellows, "Introducing an immune response into a CBR system for data mining," in *BCS ESG'96 Conference and published as Research and Development in Expert Systems XIII*. Springer-Verlag, 1996, pp. 35–42.

[67] *Proceedings of Congress on Evolutionary Computation, Part of the 2002 IEEE World Congress on Computational Intelligence held in Honolulu, HI, USA, May 12-17, 2002*. Honolulu: IEEE, May 2002.

[68] C. A. Janeway, "How the immune system recognizes invaders," *Scientific American*, vol. 269, no. 3, pp. 41–47, September 1993.

[69] N. K. Jerne, "Towards a network theory of the immune system," *Annals of Immunology (Inst. Pasteur)*, vol. 125C, pp. 373–389, 1974.

[70] J. Kennedy and R. Eberhart, *Swarm Intelligence*. Morgan Kaufmann, 2001.

[71] T. Kepler and A. Perelson, "Somatic hypermutation in B cells : An optimal control treatment," *Journal of Theoretical Biology*, vol. 164, pp. 37–64, 1993.

[72] J. W. Kim, "Integrating artificial immune algorithms for intrusion detection," Ph.D. dissertation, Department of Computer Science, University College London, 2002.

[73] T. Knight and J. Timmis, "AINE: An immunological approach to data mining," in *IEEE International Conference on Data Mining*, N. Cercone, T. Lin, and X. Wu, Eds., San Jose, CA, December 2001, pp. 297–304.

[74] T. Knight and J. Timmis, "A multi-layered immune inspired approach to data mining," in *Proceedings of the 4th International Conference on Recent Advances in Soft Computing*, A. Lotfi, J. Garibaldi, and R. John, Eds., Nottingham, UK., December 2002, pp. 266–271. URL: http://www.cs.kent.ac.uk/pubs/2002/1567

[75] T. Kohonen, *Self Organising Maps*, 2nd ed. Springer, 1997.

[76] V. Kumar and A. Gupta, "Analyzing scalability of parallel algorithms and architectures," *Journal of Parallel and Distributed Computing*, vol. 22, pp. 379–391, 1994.

[77] H. Y. Lau and V. W. Wong, "Immunologic control framework for automated material handling," pp. 57–68, in [114].

[78] D.-W. Lee, H.-B. Jun, and K.-B. Sim, "Artificial immune system for realisation of co-operative strategies and group behaviour in collective autonomous mobile robots," in *Proceedings of Fourth International Symposium on Artificial Life and Robotics*. AAAI, 1999, pp. 232–235.

[79] S.-C. Lin, W. Punch, and E. Goodman, "Coarse-grain parallel genetic algorithms: Categorization and new approach," in *Sixth IEEE Symposium on Parallel and Distributed Processing*. IEEE Computer Society Press, October 1994.

[80] S. J. Louis and G. J. E. Rawlins, "Syntactic analysis of convergence in genetic algorithms," in *Foundations of genetic algorithms 2*, L. D. Whitley, Ed. San Mateo, CA: Morgan Kaufmann, 1993, pp. 141–151. URL: citeseer.ist.psu.edu/louis92syntactic.html

[81] E. Luke, "Notes on speedup and bag-of-tasks," `http://www.cse.msstate.edu/~luke/Courses/sp04/CS4163/notes.html`, 2004.

[82] G. Marwah and L. Boggess, "Artificial immune systems for classification: Some issues," pp. 149–153, in [115].

[83] P. Matzinger, "An innate sense of danger," *Seminar in Immunology*, vol. 5, pp. 399–415, 1998.

[84] R. Medzhitov and C. A. Janeway, "Innate immunity: impact on the adaptive immune response," *Current Opinion in Immunology*, vol. 9, no. 1, pp. 4–9, February 1997.

[85] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[86] T. M. Mitchell, *Machine Learning*. Boston: WCB McGraw-Hill, 1997.

[87] N. Mitsumoto, T. Fukuda, and T. Idogaki, "Self-organising multiple robotic system," in *Proceedings of IEEE International Conference on Robotics and Automation*. Minneapolis, USA: IEEE, 1996, pp. 1614–1619.

[88] K. R. Müller, S. Mika, G. Rätsch, K. Tsuda, and B. Schölkopf, "An introduction to kernel-based learning algorithms," *IEEE Transactions on Neural Networks*, vol. 12, no. 2, pp. 181–202, March 2001.

[89] O. Nasraoui, F. Gonzalez, C. Cardona, C. Rojas, and D. Dasgupta, "A scalable artificial immune system model for dynamic unsupervised learning," pp. 219–230, in [2].

[90] O. F. Nasraoui, F. Gonzalez, and D. Dasgupta, "The fuzzy artificial immune system: Motivations, basic concepts and application to clustering and web profiling," in *International Joint Conference on Fuzzy Systems. Part of the World Congress on Computational Intelligence*. Honolulu, HI.: IEEE, 2002, pp. 711–717.

[91] M. Neal, "An artificial immune system for continuous analysis of time-varying data," pp. 76–85, in [115].

[92] M. Neal, "Meta-stable memory in an artificial immune network," pp. 168–180, in [114].

[93] G. Nicosia, V. Cutello, P. Bentley, and J. Timmis, Eds., *Proceedings of the 3rd Annual International Conference on Artificial Immune Systems (ICARIS2004)*, ser. Lecture Notes in Computer Science, no. 3239, September 2004.

[94] F. Niño, D. Gómez, and R. Vejar, "A novel immune anomaly detection technique based on negative selection," pp. 243–245, in [2].

[95] E. Noda, A. L. V. Coelho, I. L. M. Ricarte, A. Yamakami, and A. A. Freitas, "Devising adaptive migration policies for cooperative distributed genetic algorithms," in *Proceedings of 2002 IEEE International Conference on Systems, Man and Cybernetics (SMC-2002)*, 2002.

[96] G. J. V. Nossal, "Life, death, and the immune system," *Scientific American*, vol. 269, no. 3, pp. 21–30, September 1993.

[97] G. J. V. Nossal, "Negative selection of lymphocytes," *Cell*, vol. 76, pp. 229–239, January 28 1994.

[98] M. Oprea and S. Forrest, "Simulated evolution of antibody gene libraries under pathogen selection," in *Proceedings of the 1998 IEEE International Conference on Systems, Man and Cybernetics*, 1998.

[99] D. Pelleg and A. Moore, "Accelerating exact k-means algorithms with geometric reasoning," in *Proceedings of the 1999 Conference on Knowledge Discovery in Databases (KDD99)*, 1999.

[100] A. Perelson, "Immune network theory," *Immunological Review*, vol. 110, pp. 5–36, 1989.

[101] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA: Morgan Kaufmann, 1993.

[102] J. A. Rice, *Mathematical Statistics and Data Analysis*, 2nd ed. Belmont, California: Duxbury Press, 1995.

[103] I. M. Roitt and P. J. Delves, *Roitt's Essential Immunology*, 10th ed. Oxford: Blackwell Science, 2001.

[104] J. M. Slack, *From Egg to Embryo*. Cambridge University Press, 1991.

[105] V. Slavov and N. Nikolaev, "Immune network dynamics for inductive problem solving," in *Proceedings of Parallel Problem Solving from Nature - PPSN V: 5th International Conference*, ser. Lecture Notes in Computer Science, A. E. Eiben, T. Back, M. Schoenauer, and H. P. Schwefel, Eds., no. 1498. Springer-Verlag, September 1998, pp. 712–721.

[106] M. Snir and S. Otto, *MPI-The Complete Reference: The MPI Core*. MIT Press, 1998.

[107] J. Sprent, "T and B memory cells," *Cell*, vol. 76, pp. 315–322, January 28 1994.

[108] S. Stepney, R. Smith, J. Timmis, and A. Tyrrell, "Towards a conceptual framework for artificial immune systems," pp. 53–64, in [93].

[109] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. MIT Press, 1998.

[110] R. Tanese, "Parallel genetic algorithm for a hypercube," in *Proceedings of the Second International Conference on Genetic Algorithms*, J. J. Grefenstette, Ed., 1987, pp. 177–183.

[111] R. Tanese, "Distributed genetic algorithms," in *Proceedings of the Third International Conference on Genetic Algorithms*, J. D. Schaffer, Ed., 1989, pp. 434–439.

[112] D. W. Taylor and D. W. Corne, "An investigation of the negative selection algorithm for fault detection in refrigeration systems," pp. 34–45, in [114].

[113] J. Timmis, "Artificial immune systems: A novel data analysis technique inspired by the immune network theory," Ph.D. dissertation, Department of Computer Science, University of Wales, Aberystwyth. Ceredigion. Wales., August 2000. URL: http://www.cs.kent.ac.uk/pubs/2000/1102

[114] J. Timmis, P. Bentley, and E. Hart, Eds., *Proceedings of the 2nd International Conference on Artificial Immune Systems (ICARIS2003)*, ser. Lecture Notes in Computer Science, no. 2787. Springer-Verlag, September 2003.

[115] J. Timmis and P. Bentley, Eds., *1st International Conference on Artificial Immune Systems*. University of Kent at Canterbury: Unversity of Kent at Canterbury Printing Unit, September 2002.

[116] J. Timmis, M. Neal, and J. Hunt, "An artificial immune system for data analysis," *Biosystems*, vol. 55, no. 1/3, pp. 143–150, 2000.

[117] J. Timmis and M. Neal, "A resource limited artificial immune system for data analysis," *Knowledge Based Systems*, vol. 14, no. 3-4, pp. 121–130, June 2001. URL: http://www.cs.kent.ac.uk/pubs/2001/1207

[118] S. Tonegawa, "Somatic generation of antibody diversity," *Nature*, vol. 302, no. 14, pp. 575–581, 1983.

[119] S. Tonegawa, "The molecules of the immune system," *Scientific American*, vol. 253, no. 4, pp. 104–131, October 1985.

[120] A. Ultsch, *Kohonen Maps*. Elsevier Science, 1999.

[121] V. Vapnik, *The Nature of Statistical Learning Theory*. New York: Springer, 1995.

[122] F. Varela, A. Coutinho, B. Dupire, and N. Vaz, "Cognitive networks : Immune and neural and otherwise," *Theoretical Immunology : Part Two, SFI Studies in the Sciences of Complexity*, vol. 2, pp. 359–371, 1988.

[123] M. Villalobos-Arias, C. A. C. Coello, and O. Hernández-Lerma, "Convergence analysis of a multiobjective artificial immune system algorithm," pp. 226–235, in [93].

[124] A. Watkins, "AIRS: A resource limited artificial immune classifier," Master's thesis, Department of Computer Science, College of Engineering. Mississippi State University, 2001. URL: http://nt.library.msstate.edu/etd/show.asp?etd=etd-11052001-102048

[125] A. Watkins, X. Bi, and A. Phadke, "Parallelizing an immune-inspired algorithm for efficient pattern recognition," in *Intelligent Engineering Systems through Artificial Neural Networks: Smart Engineering System Design: Neural Networks, Fuzzy Logic, Evolutionary Programming, Complex Systems and Artificial Life*, C. Dagli, A. Buczak, J. Ghosh, M. Embrechts, and O. Ersoy, Eds. New York: ASME Press, November 2003, vol. 13, pp. 225–230.

[126] A. Watkins and L. Boggess, "A new classifier based on resource limited artificial immune systems," pp. 1546–1551, in [67].

[127] A. Watkins and L. Boggess, "A resource limited artificial immune classifier," pp. 926–931, in [67].

[128] A. Watkins and J. Timmis, "Artificial immune recognition system (AIRS): Revisions and refinements," pp. 173–181, in [115].

[129] A. Watkins and J. Timmis, "Exploiting parallelism inherent in AIRS, an artificial immune classifier," pp. 427–438, in [93].

[130] A. Watkins, J. Timmis, and L. Boggess, "Artificial immune recognition system (AIRS): An immune-inspired supervised machine learning algorithm," *Genetic Programming and Evolvable Machines*, vol. 5, no. 3, pp. 291–317, September 2004.

[131] L. Wolpert, *Principles of Development*. Oxford University Press, 1998.

Appendix A

Overview of the AIRS algorithm

This appendix presents an overview of the AIRS algorithm.[1] A.1 defines some of the key terms and concepts important to the understanding of the algorithm, and A.2 provides a somewhat formal tour of the training routine of the algorithm.

A.1 Definitions

This subsection presents definitions of the key terms and concepts used throughout the rest of this thesis, particularly as they apply to the AIRS algorithm.

- *affinity*: a measure of "closeness" or similarity between two *antibodies* or *antigens*. In the current implementation, this value is guaranteed to be between 0 and 1 inclusively and is calculated simply as the Euclidean distance of the two objects' *feature vectors*. Thus, small affinity values indicate strong affinity.

[1] This overview is taken directly from [124].

- *affinity threshold (AT)*: the average *affinity* value among all of the *antigens* in the *training set* or among a selected subset of these training antigens.
- *affinity threshold scalar (ATS)*: a value between 0 and 1 that, when multiplied by the *affinity threshold*, provides a cut-off value for memory cell replacement in the *AIRS* training routine.
- *antibody*: a *feature vector* coupled with its associated *output* or *class*; the feature vector-output combination is referred to as an antibody when it is part of an *ARB* or *memory cell*.
- *antigen*: this is the same in representation as an *antibody*; however, the feature vector-class combination is referred to as an antigen when it is being presented to the *ARBS* for stimulation and/or response.
- *Artificial Immune Recognition System (AIRS)*: a classification algorithm inspired by natural immune systems.
- *Artificial Recognition Ball (ARB)*: also known as a *B-cell*. It consists of an *antibody*, a count of the number of *resources* held by the cell, and the current *stimulation value* of the cell.
- *B Cell*: in this thesis, more commonly referred to as an *Artificial Recognition Ball*.
- *candidate Memory Cell*: the *antibody* of an *ARB*, of the same *class* as the training *antigen*, which was the most stimulated after exposure to the given antigen.

- *class*: the category of a given *feature vector*. This is also referred to as the *output* of a cell.

- *clonal rate*: an integer value used to determine the number of mutated clones a given *ARB* is allowed to attempt to produce. In the current implementation, a selected ARB is allowed to produce up to *clonal rate* * *stimulation value* mutated clones after responding to a given *antigen*. This product is also used in assigning *resources* to an ARB. Therefore, the clonal rate serves a dual-role as resource allocation factor and clonal mutation factor for the cell population.

- *established Memory Cell*: the *antibody* of an *ARB* which has survived competition for resources and was the most stimulated to a given training *antigen* and has been added to the evolving set of *memory cells*.

- *feature vector*: one instance of data represented as a sequence of values. Each position in the sequence represents a different feature associated with the data, and each feature has its own range of legitimate values.

- *hyper-mutation rate*: an integer value used to determine the number of mutated clones a given *memory cell* is allowed to inject into the cell population. In the initial implementation, the selected memory cell injects at least *hyper-mutation rate* * *clonal rate* * *stimulation value* mutated clones into the cell population at the time of *antigen* introduction.

- *k nearest neighbor (KNN)*: a classification scheme in which the response of the classifier to a previously unseen item is determined by a majority vote among the k closest data points. For the *AIRS* algorithm, the k closest data points are in actuality the k most stimulated *memory cells* to a given test *antigen*.

- *k value*: the parameter which indicates how many *memory cells* should be used to determine the classification of a given test item. (see *k nearest neighbor* for more details)

- *memory cell (mc)*: the *antibody* of an *ARB* which was the most stimulated by a given training *antigen* at the end of exposure to that antigen. It is used for hyper-mutation in response to incoming training antigens (see *hyper-mutation rate*). An mc can be replaced, however. This occurs only when a *candidate mc* is more stimulated to a given training antigen than the most stimulated *established mc* and the affinity between the established mc and the candidate mc is less than the product of the *Affinity Threshold* and the *Affinity Threshold Scalar*.

- *mutation rate*: a parameter between 0 and 1 that indicates the probability that any given feature (or the output) of an *ARB* will be mutated.

- *output*: the classification category associated with a cell. Same as the *class* of the feature vector corresponding to the cell.

- *resources*: a parameter which limits the number of *ARB*s allowed in the system. Each ARB is allocated a number of resources based on its *stimulation value* and the *clonal rate*. The total number of system wide resources is set to a certain limit. If more resources are consumed than are allowed to exist in the system, then resources are removed from the least stimulated ARBs until the number of resources in the system returns to the number allowed. If all of a given ARB's resources are removed, then that ARB is removed from the cell population.

- *seed cell*: an *antibody*, drawn from the *training set*, used to initialize *Memory Cell* and *ARB* populations at the beginning of training.

- *stimulation function*: a function used to measure the response of an *ARB* to an *antigen* or to another ARB. In the current formulation of the *AIRS* classifier, this function should return a value between 0 and 1 inclusively. For the implementation of *AIRS* presented in this study, the stimulation function is inversely proportional to the Euclidean distance between the *feature vector*s of the *ARB* and the *antigen*.

- *stimulation value*: the value returned by the *stimulation function*.

- *stimulation threshold*: a parameter between 0 and 1 used as a stopping criterion for the training on a specific *antigen*. For the initial implementation, only when the average *stimulation value* of the *ARB*s of

each class is above the stimulation threshold does training in reaction to the particular antigen stop.

- *test set*: the collection of *antigens* used to evaluate the classification performance of the trained *AIRS* classifier.

- *training set*: the collection of *antigens* used to train the *AIRS* classifier.

A.2 Tour of the Algorithm

This section presents a tour of the AIRS algorithm. In particular, this section presents an overview of the primary routines, methods, and equations used in the training and building of an immune-system based classifier. There are four primary stages involved in the AIRS algorithm. The first stage is data normalization and initialization. The second stage is memory cell identification and ARB generation. The third stage is competition for resources in the development of a candidate memory cell. The final stage of the training algorithm is the potential introduction of the candidate memory cell into the set of established memory cells.

For this discussion, let us establish the following notational conventions:

- Let MC represent the set of memory cells and mc represent an individual member of this set.

- Let $ag.c$ represent the class of a given antigen, ag, where $ag.c \in C = \{1, 2, \ldots nc\}$ and nc is the number of classes in the data set.

- Let $mc.c$ represent the class of a given memory cell, mc, where $mc.c \in C = \{1, 2, \ldots nc\}$.
- Define $MC_c \subseteq MC = \{MC_1 \cup MC_2 \cup \ldots MC_{nc}\}$ and $mc \in MC_c$ **iff** $mc.c \equiv c$.
- Let $ag.f$ and $mc.f$ represent the feature vector of a given antigen and memory cell, ag and mc, respectively. Let $ag.f_i$ represent the value of the ith feature in $ag.f$ and $mc.f_i$ the value of the ith value of $mc.f$.
- Let AB represent the set of ARBs, or the population of existing cells, and MU represent a set of mutated clones of ARBs. Furthermore, let ab represent a single ARB where $ab \in AB$.
- Let $ab.c$ represent the class of a given ARB, ab, where $ab.c \in C = \{1, 2, \ldots nc\}$.
- Define $AB_c \subseteq AB = \{AB_1 \cup AB_2 \cup \ldots AB_{nc}\}$, and $ab \in AB_c$ iff $ab.c \equiv c$.
- Let $ab.stim$ represent the stimulation level of the ARB ab.
- Let $ab.resources$ represent the number of resources held by the ARB ab.
- Let $TotalNumResources$ represent the total number of system wide resources allowed.

A.2.1 Initialization

The first stage of the algorithm, initialization, can primarily be thought of as a data pre-processing stage combined with a parameter discovery stage. During initialization, first all items in the data set are normalized such that the Euclidean distance between the feature vectors of any two items is in the range of [0,1].[2] This can be performed through a variety of methods and could also be performed as a true pre-processing stage before the algorithm begins. It is important to note that, while for the current investigation Euclidean distance is the primary metric of both affinity and stimulation, other functions could be employed as well. What is important about this normalization is only that the range of possible reactions from cell-to-cell interaction remains within the range of [0,1]. After normalization, the affinity threshold is calculated. The affinity threshold is the average affinity value over all training data items' feature vectors. The affinity threshold is calculated as described in equation (A.2.1) below:

$$\text{affinity threshold} = \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} \text{affinity}(ag_i, ag_j)}{\frac{n(n-1)}{2}} \tag{A.2.1}$$

where n is the number of training data items (antigens) in question, ag_i and ag_j are the ith and jth training antigen in the antigen training vector, and affinity(x,y) returns the Euclidean distance between the two antigens' feature vectors.

[2]Euclidean distance was chosen as an initial starting place for this prototype as it has been used in many standard machine learning algorithms.

The final step in initialization is the seeding of the memory cells and initial ARB population. This is done by randomly choosing 0 or more antigens from the set of training vector to be added to the set of memory cells and to the set of ARBs.

A.2.2 Memory Cell Identification and ARB Generation

Once initialization is complete, training proceeds as a one-shot incremental algorithm. That is, each element of the training data is presented to the AIRS learning algorithm exactly once. The first step of this stage of the algorithm is memory cell identification and ARB generation. Given a specific training antigen, ag, find the memory cell, mc_{match}, that has the following property:

$$mc_{match} = \arg max_{mc \in MC_{ag.c}} stimulation(ag, mc) \tag{A.2.2}$$

where $stimulation(x, y)$ is defined as in equation (A.2.3) below:

$$stimulation(x, y) = 1 - \text{affinity}(x, y) \tag{A.2.3}$$

If $MC_{ag.c} \equiv \oslash$, then $mc_{match} \leftarrow ag$ and $MC_{ag.c} \leftarrow MC_{ag.c} \cup ag$. That is, if the set of memory cells of the same classification as the antigen is empty, then add the antigen to the set of memory cells and denote this newly added memory cell as the match memory cell, mc_{match}. It should be noted here that while the stimulation

function for the current work relies solely on Euclidean distance, this need not necessarily be the case.

Once mc_{match} has been identified, this memory cell is used to generate new ARBs to be placed into the population of (possibly) pre-existing ARBs (*i.e.,* those ARBs left in the system from exposure to previous antigens). This is done through the method shown in Figure A.2.1, where the function $makeARB(x)$ returns an ARB with x as the antibody of this ARB and where $mutate(x, b)$ is defined in Figure A.2.2. In Figure A.2.2, the function $drandom()$ returns a random value

$MU \leftarrow \oslash$
$MU \leftarrow MU \cup makeARB(mc_{match})$
$stim \leftarrow stimulation(ag, mc_{match})$
$NumClones \leftarrow hyper\_clonal\_rate * clonal\_rate * stim$
while ($\mid MU \mid < NumClones$)
do
  $mut \leftarrow false$
  $mc_{clone} \leftarrow mc_{match}$
  $mc_{clone} \leftarrow mutate(mc_{clone}, mut)$
  if($mut \equiv true$)
    $MU \leftarrow MU \cup makeARB(mc_{clone})$
done
$AB \leftarrow AB \cup MU$

Figure A.2.1: Hyper-Mutation for ARB Generation

in the range [0,1] and ($lrandom()$ mod nc) returns a random value in the range $\{0,nc\}$.

```
mutate(x, b)
{
  foreach(x.f_i in x.f)
  do
    change ← drandom()
    change_to ← drandom()
    if(change < mutation_rate)
      x.f_i ← change_to * normalization_value
      b ← true
  done
  if(b ≡ true)
    change ← drandom()
    change_to ← (lrandom() mod nc)
    if(change < mutation_rate)
      x.c ← change_to
  return x
}
```

Figure A.2.2: Mutation Routine

A.2.3 Competition for Resources and Development of a Candidate Memory Cell

At this point a set of ARBs (AB) exists which includes mc_{match}, mutations from mc_{match}, and (possibly) remnant ARBs from responses to previously encountered antigens. Recall that the AIRS algorithm is a one-shot algorithm, so while the discussion has been divided into separate stages, only one antigen goes through this entire process at time (with the obvious exception being the initialization stage which takes place over the entire data set before training begins). The goal of the next portion of the algorithm is to develop a candidate memory cell which is most successful in correctly classifying a given antigen, ag. This is done primarily through three mechanisms. The first mechanism is through the

competition for system wide resources. Following the methods first outlined by [116] and more fully realized by [117], resources are allocated to a given ARB based on its normalized stimulation value, which is used as an indication of its fitness as a recognizer of ag. The second mechanism is through the use of mutation for diversification and shape-space exploration. The third mechanism is the use of an average stimulation threshold as a criterion for determining when to stop training on ag.

Similar to principles involved in genetic algorithms, the AIRS algorithm employs a concept of fitness for survival of individuals within the ARB population. Survival of a given ARB is determined in a two-fold, interrelated manner. First, each ARB in the population AB is presented with the antigen ag to determine the ARB's stimulation level. This stimulation is then normalized across the ARB population based on both the raw stimulation level and the class of the given ARB ($ab.c$). Based on this normalized stimulation value, each $ab \in AB$ is allocated a finite number of resources. If this allocation of resources would result in more resources being allocated across the population than allowed, then resources are removed from the weakest (least stimulated) ARBs until the total number of resources in the system returns to the number of resources allowed. Those ARBs which have zero resources are removed from the ARB population. This process is formalized in Figure A.2.3.

Two key aspects of this resource allocation routine for the initial formulation of the AIRS algorithm are noted here. First, the stimulation value of an ARB is not only determined by the stimulation function in equation A.2.3 but is also

$minStim \leftarrow MAX$
$maxStim \leftarrow MIN$
foreach($ab \in AB$)
do
  $stim \leftarrow stimulation(ag, ab)$
  if ($stim < minStim$)
    $minStim \leftarrow stim$
  if ($stim > maxStim$)
    $maxStim \leftarrow stim$
  $ab.stim \leftarrow stim$
done
foreach($ab \in AB$)
do
  if($ab.c \equiv ag.c$)
    $ab.stim \leftarrow \frac{ab.stim - minStim}{maxStim - minStim}$
  else
    $ab.stim \leftarrow 1 - \frac{ab.stim - minStim}{maxStim - minStim}$
  $ab.resources \leftarrow ab.stim * clonal\_rate$
done
$i \leftarrow 1$
while($i \leq nc$)
do
  $resAlloc \leftarrow \sum_{j=1}^{|AB_i|} ab_j.resources, ab_j \in AB_i$
  if($i \equiv ag.c$)
    $NumResAllowed \leftarrow \frac{TotalNumResources}{2}$
  else
    $NumResAllowed \leftarrow \frac{TotalNumResources}{2*(nc-1)}$
  while($resAlloc > NumResAllowed$)
  do
    $NumResRemove \leftarrow resAlloc - NumResAllowed$
    $ab_{remove} \leftarrow argmin_{ab \in AB_i}(ab.stim)$
    if($ab_{remove}.resources \leq NumResRemove$)
      $AB_i \leftarrow AB_i - ab_{remove}$
      $resAlloc \leftarrow resAlloc - ab_{remove}.resources$
    else
      $ab_{remove}.resources \leftarrow ab_{remove}.resources - NumResRemove$
      $resAlloc \leftarrow resAlloc - NumResRemove$
  done
  $i \leftarrow i + 1$
done

Figure A.2.3: Stimulation, Resource Allocation, and ARB Removal

based on the class of the ARB. The stimulation calculation method outlined in Figure A.2.3 provides reinforcement both for those ARBs of the same class as ag that are highly stimulated by ag and for those ARBs that are of a different class from ag that do not exhibit a strong positive reaction to ag. Second, the distribution of resources is also based on the class of the ARB. This is done to provide additional reinforcement for those ARBs of the same class as ag without losing the potentially positive qualities of the remaining ARBs for reaction to future antigens.

At this point in the algorithm, the ARB population AB consists of only those ARBs that were most stimulated by the given antigen, ag, or more specifically, AB now consists of those ARBs that were able to successfully compete for resources. The algorithm continues first by determining if the ARBs in AB were stimulated enough by ag to stop training on this item. This is done by defining a vector $\vec{s}$ that is nc in length to contain the average stimulation value for each class subset of AB. That is:

$$s_i \leftarrow \frac{\sum_{j=1}^{|AB_i|} ab_j.stim}{\mid AB_i \mid}, ab_j \in AB_i \tag{A.2.4}$$

The stopping criterion is reached iff $s_i \geq stimulation\_threshold$ for all elements in $\vec{s} = \{s_1, s_2, \ldots s_{nc}\}$.

Regardless of whether the stopping criterion is met or not, the algorithm proceeds by allowing each ARB in AB the opportunity to produce mutated offspring. While this adding of mutated offspring is similar to the method outlined

in Figure A.2.1, there are a few differences. This modified mutation generation routine is presented in Figure A.2.4.

```
MU ← ∅
foreach(ab ∈ AB)
do
  rd ← drandom()
  if(ab.stim > rd)
    NumClones ← ab.stim * clonal_rate
    i ← 1
    while(i ≤ NumClones)
    do
      mut ← false
      ab_clone ← ab
      ab_clone ← mutate(ab_clone, mut)
      if(mut ≡ true)
        MU ← MU ∪ ab_clone
      i ← i + 1
    done
done
AB ← AB ∪ MU
```

Figure A.2.4: Mutation of Surviving ARB

After allowing each surviving ARB the opportunity to produce mutated offspring, the stopping criterion is examined. If the stopping criterion is met, then training on this one antigen stops. If the stopping criterion has not been met, then this entire process, beginning with the method outlined in Figure A.2.3, is repeated until the stopping criterion is met. The only exception to this repetition is that on every pass through this portion of the algorithm, except the first pass already discussed, if the stopping criterion is met after the stimulation and resource allocation phase, then the production of mutated offspring is not performed. Once the stopping criterion has been met, then the candidate memory

cell is chosen. The candidate memory cell, $mc_{candidate}$, is the feature vector and class of the ARB that existed in the system before the most recent round of mutations that was the most stimulated ARB of the same class as the training antigen ag.

A.2.4 Memory Cell Introduction

The final stage in the training routine is the potential introduction of the just-developed candidate memory cell, $mc_{candidate}$, into the set of existing memory cells MC. It is during this stage that the affinity threshold calculated during initialization becomes critical as it dictates whether the $mc_{candidate}$ replaces mc_{match} that was previously identified. The candidate memory cell is added to the set of memory cells only if it is more stimulated by the training antigen, ag, than mc_{match}, where stimulation is defined as in equation (A.2.3). If this test is passed, then if the affinity between $mc_{candidate}$ and mc_{match} is less than the product of the affinity threshold and the affinity threshold scalar, then $mc_{candidate}$ replaces mc_{match} in the set of memory cells. This memory cell introduction method is presented in figure A.2.5.

Once the candidate memory cell has been evaluated for addition into the set of established memory cells, training on this one antigen is complete. The next antigen in the training set is then selected, and the training process proceeds with memory cell identification and ARB generation. This process continues until all antigens have been presented to the system.

$CandStim \leftarrow stimulation(ag, mc_{candidate})$
$MatchStim \leftarrow stimulation(ag, mc_{match})$
$CellAff \leftarrow affinity(mc_{candidate}, mc_{match})$
if($CandStim > MatchStim$)
  if($CellAff < AT * ATS$)
    $MC \leftarrow MC - mc_{match}$
  $MC \leftarrow MC \cup mc_{candidate}$

Figure A.2.5: Memory Cell Introduction

A.2.5 Classification

After training has completed, the evolved memory cells are available for use for classification. The classification is performed in a k-nearest neighbor approach. Each memory cell is iteratively presented with each data item for stimulation. The system's classification of a data item is determined by using a majority vote of the outputs of the k most stimulated memory cells.

Appendix B

Statistical Note

B.1 t-test

In order to test for the statistical significance between two results, we use Student's t-test throghout this thesis [102, chap11, pp387-441]. The hypothesis that we are testing throughout is that the mean value of two results are the same. We use the two-tailed t-test to determine if we can accept or reject this hypothesis at a given significance level. Or, to put it another way, we determine the probablity (α) of rejecting this hypothesis when, in fact, it is true. For all of our tests, we assume that the mean and variance of the samples are unknown. Given these assumptions, we calcluate the t-statistic with the following formula:

$$T_f = \frac{\bar{X}_1 - \bar{X}_2}{s_p \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \tag{B.1.1}$$

where $\bar{X}_i$ is the sample mean, s_p is the square root of the pooled sample variance, and n_i is the sample size. The pooled sample variance is calculated by:

$$s_p^2 = \frac{(n_1 - 1)s_1^2 + (n_2 - 1)s_2^2}{n_1 + n_2 - 2} \tag{B.1.2}$$

where s_i^2 is the sample vairance. The degrees of freedom (f) for this variable T is calculated by $x_1 + x_2 - 2$. Once we have f caculated, we can use this information to look up the t-distribution value to determine the the rejection region for our hypothesis that the two means are equal. We will reject our hypothiesis if:

$$|T| \geq t_{f,1-\frac{\alpha}{2}} \tag{B.1.3}$$

Otherwise, we can accept our hypothesis at the $(1 - \alpha)\%$ significance level. Alternatively, we can us the value of T and f to determine the p value of this statistic. The p value indicates the probability that the two means are the same. If $p < \alpha$, then we decide that this probabilty is not sufficient for the difference between the means to be based only on chance, and thus there is a statistically significant difference in the means.

B.2 Parallel Speedup and Efficiency

Since both speedup and efficiency are ratios of times from two independent distributions, we want to provide a means of determining the mean and standard deviaions of these ratios in order to report error bars for these measurements. In

order to calculate the mean value for the speedup we simply find the mean of the pairwise combination of the sample ratios:

$$\bar{S} = \frac{1}{n_1 * n_p} \sum_{i=1}^{n_1} \sum_{j=1}^{n_p} \frac{t_{1,i}}{t_{p,j}} \tag{B.2.1}$$

where n_1 and n_p are the number of runs on a single processors and the number of runs with the same parameters on multiple processors, respectively, $t_{1,i}$ is the serial time for the ith run on one processor, and $t_{p,j}$ is the parallel time for the jth run on multiple processors. The mean efficiency can be calculated in a similar manner:

$$\bar{E_p} = \frac{1}{n_1 * n_p} \sum_{i=1}^{n_1} \sum_{j=1}^{n_p} \frac{t_{1,i}}{np * t_{p,j}} \tag{B.2.2}$$

where np is the number of processors used. We also can use these pairwise ratios to calculate standard deviation in the usual way. For example,

$$\sqrt{\frac{1}{(n_1 * n_p) - 1} \sum_{i=1}^{n_1} \sum_{j=1}^{n_p} (\frac{t_{1,i}}{t_{p,j}} - \bar{S})^2} \tag{B.2.3}$$

gives the standard deviation for speedup.

Appendix C

Parallel CLONALG Results

This appendix presents results for experiments with the parallel version of CLONALG.

Table C.0.1: Parallel CLONALG: Runtimes when Varying N

np⇒	1	2	4	8
N⇓	Time	Time	Time	Time
10	11.86(0.03)	6.39(0.08)	3.68(0.01)	1.85(0.07)
20	15.07(0.05)	7.74(0.07)	3.79(0.02)	2.28(0.05)
40	23.21(0.02)	10.03(0.14)	5.75(0.09)	2.81(0.08)
80	33.49(0.02)	16.45(0.12)	9.17(0.15)	4.73(0.06)
160	57.58(0.03)	31.08(0.08)	16.20(0.22)	7.64(0.21)

Table C.0.2: Parallel CLONALG: Generations to Converge when Varying N

np⇒	1	2	4	8
N⇓	Gens	Gens	Gens	Gens
10	72(0)	75(0)	87(0)	82(0)
20	44(0)	44(0)	43(0)	51(0)
40	32(0)	27(0)	30(0)	29(0)
80	22(0)	21(0)	23(0)	24(0)
160	18(0)	19(0)	19(0)	18(0)

Table C.0.3: Parallel CLONALG: Runtimes when Varying $n1$

np⇒	1	2	4	8
n1⇓	Time	Time	Time	Time
2	9.64(0.02)	4.89(0.03)	2.56(0.09)	1.60(0.05)
4	16.20(0.02)	8.01(0.05)	4.09(0.06)	1.91(0.07)
8	23.21(0.02)	10.03(0.14)	5.75(0.09)	2.81(0.08)
16	31.68(0.02)	14.70(0.12)	8.16(0.17)	4.53(0.07)
32	49.84(0.02)	23.57(0.31)	12.80(0.36)	6.24(0.07)

Table C.0.4: Parallel CLONALG: Generations to Converge when Varying $n1$

np⇒	1	2	4	8
n1⇓	Gens	Gens	Gens	Gens
2	31(0)	31(0)	30(0)	36(0)
4	33(0)	32(0)	32(0)	29(0)
8	32(0)	27(0)	30(0)	29(0)
16	31(0)	27(0)	29(0)	33(0)
32	33(0)	28(0)	31(0)	29(0)

Table C.0.5: Parallel CLONALG: Runtimes when Varying $n2$

np⇒	1	2	4	8
n2⇓	Time	Time	Time	Time
0	27.34(0.02)	10.34(0.03)	5.31(0.09)	3.11(0.06)
1	23.19(0.03)	11.44(0.05)	5.89(0.11)	2.78(0.13)
2	23.21(0.02)	10.03(0.14)	5.75(0.09)	2.81(0.08)
4	22.51(0.02)	11.37(0.04)	5.85(0.08)	3.04(0.07)
8	24.64(0.03)	11.03(0.04)	5.64(0.05)	2.80(0.07)

Table C.0.6: Parallel CLONALG: Generations to Converge when Varying $n2$

np⇒	1	2	4	8
n2⇓	Gens	Gens	Gens	Gens
0	38(0)	28(0)	28(0)	32(0)
1	32(0)	31(0)	31(0)	28(0)
2	32(0)	27(0)	30(0)	29(0)
4	31(0)	31(0)	30(0)	31(0)
8	34(0)	30(0)	30(0)	29(0)

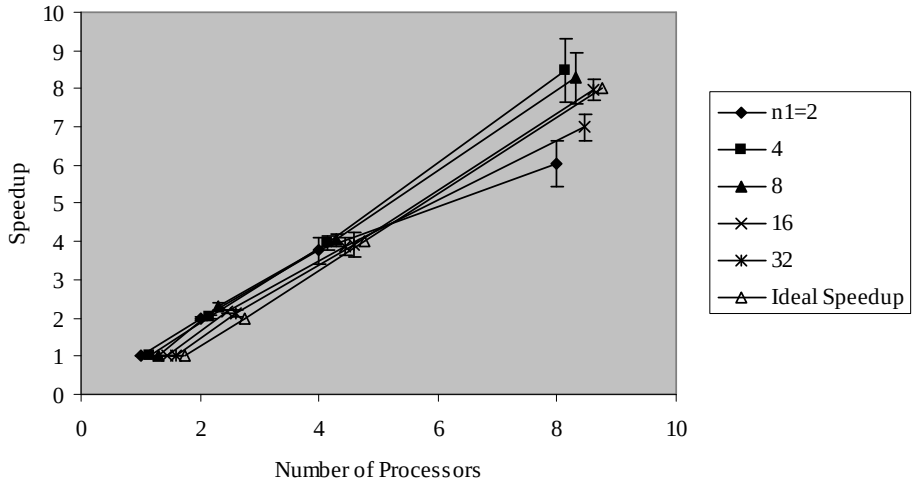

Figure C.0.1: Speedup of CLONALG when varying $n1$ (x-axis offset applied for visual clarity)

Table C.0.7: Parallel CLONALG: Speedup when Varying N

np⇒	1	2	4	8
N⇓	S	S	S	S
10	1.00(0.00)	1.86(0.02)	3.23(0.01)	6.43(0.23)
20	1.00(0.00)	1.95(0.02)	3.98(0.03)	6.62(0.14)
40	1.00(0.00)	2.31(0.03)	4.04(0.06)	8.28(0.22)
80	1.00(0.00)	2.04(0.01)	3.65(0.06)	7.08(0.09)
160	1.00(0.00)	1.85(0.00)	3.56(0.05)	7.54(0.19)

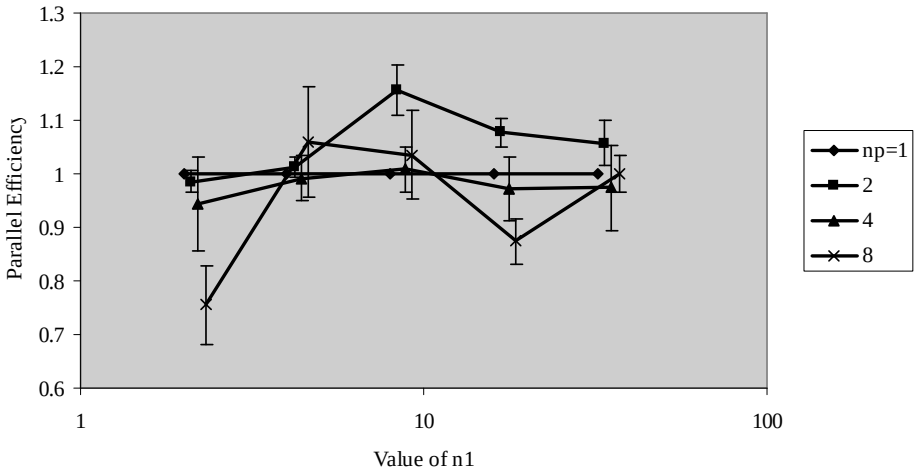

Figure C.0.2: Parallel efficiency of CLONALG when varying $n1$ (x-axis offset applied for visual clarity)

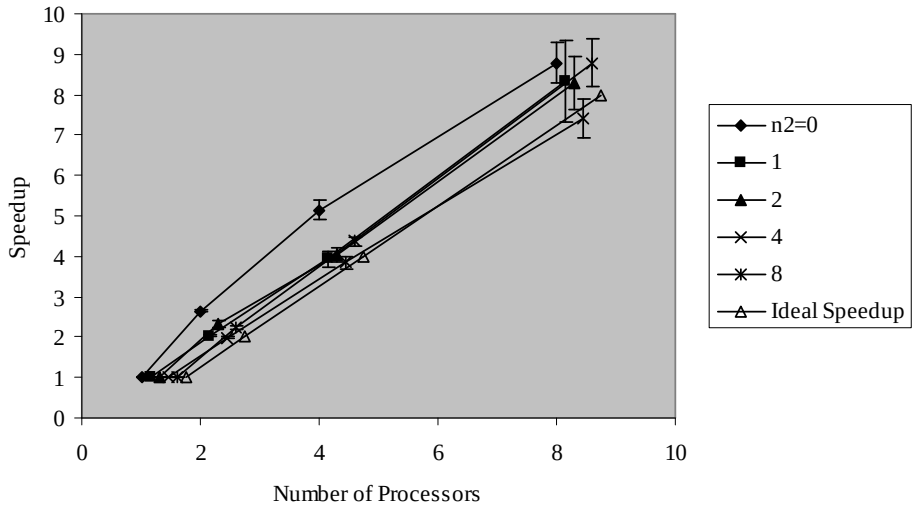

Figure C.0.3: Speedup of CLONALG when varying $n2$ (x-axis offset applied for visual clarity)

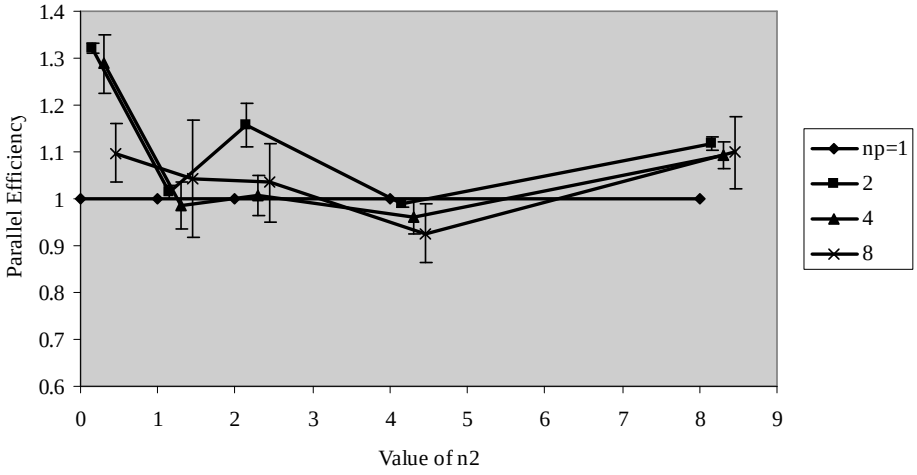

Figure C.0.4: Parallel efficiency of CLONALG when varying $n2$ (x-axis offset applied for visual clarity)

Table C.0.8: Parallel CLONALG: Parallel Efficiency when Varying N

np⇒	1	2	4	8
N⇓	E_p	E_p	E_p	E_p
10	1.00(0.00)	0.93(0.01)	0.81(0.00)	0.80(0.03)
20	1.00(0.00)	0.97(0.01)	1.00(0.01)	0.83(0.02)
40	1.00(0.00)	1.16(0.02)	1.01(0.01)	1.04(0.03)
80	1.00(0.00)	1.02(0.01)	0.91(0.01)	0.88(0.01)
160	1.00(0.00)	0.93(0.00)	0.89(0.01)	0.94(0.02)

Table C.0.9: Parallel CLONALG: Speedup when Varying $n1$

np⇒	1	2	4	8
n1⇓	S	S	S	S
2	1.00(0.00)	1.97(0.01)	3.77(0.12)	6.04(0.19)
4	1.00(0.00)	2.02(0.01)	3.97(0.06)	8.47(0.28)
8	1.00(0.00)	2.31(0.03)	4.04(0.06)	8.28(0.22)
16	1.00(0.00)	2.15(0.02)	3.88(0.08)	6.99(0.11)
32	1.00(0.00)	2.11(0.03)	3.90(0.11)	7.99(0.09)

Table C.0.10: Parallel CLONALG: Parallel Efficiency when Varying $n1$

np⇒	1	2	4	8
n1⇓	E_p	E_p	E_p	E_p
2	1.00(0.00)	0.99(0.01)	0.94(0.03)	0.75(0.02)
4	1.00(0.00)	1.01(0.01)	0.99(0.01)	1.06(0.03)
8	1.00(0.00)	1.16(0.02)	1.01(0.01)	1.04(0.03)
16	1.00(0.00)	1.08(0.01)	0.97(0.02)	0.87(0.01)
32	1.00(0.00)	1.06(0.01)	0.97(0.03)	1.00(0.01)

Table C.0.11: Parallel CLONALG: Speedup when Varying $n2$

np⇒	1	2	4	8
n2⇓	S	S	S	S
0	1.00(0.00)	2.64(0.01)	5.15(0.08)	8.78(0.17)
1	1.00(0.00)	2.03(0.01)	3.94(0.07)	8.35(0.33)
2	1.00(0.00)	2.31(0.03)	4.04(0.06)	8.28(0.22)
4	1.00(0.00)	1.98(0.01)	3.85(0.05)	7.41(0.16)
8	1.00(0.00)	2.23(0.01)	4.37(0.04)	8.79(0.20)

Table C.0.12: Parallel CLONALG: Parallel Efficiency when Varying $n2$

np⇒	1	2	4	8
n2⇓	E_p	E_p	E_p	E_p
0	1.00(0.00)	1.32(0.00)	1.29(0.02)	1.10(0.02)
1	1.00(0.00)	1.01(0.00)	0.99(0.02)	1.04(0.04)
2	1.00(0.00)	1.16(0.02)	1.01(0.01)	1.04(0.03)
4	1.00(0.00)	0.99(0.00)	0.96(0.01)	0.93(0.02)
8	1.00(0.00)	1.12(0.00)	1.09(0.01)	1.10(0.03)

Table C.0.13: Parallel CLONALG: Run Times when Varying the Number of Input Vectors

np→	1	2	4	8	16
M↓	time (s)	time (s)	time (s)	time (s)	time (s)
8	41.70(0.05)	21.33(0.28)	10.94(0.17)	6.09(0.10)	
16	94.31(0.06)	42.74(0.10)	21.70(0.17)	10.84(0.12)	5.53(0.10)
32	187.85(0.08)	95.95(0.22)	48.34(0.15)	24.26(0.36)	12.10(0.10)
64	377.83(0.10)	192.01(0.39)	96.14(0.22)	48.33(0.28)	24.35(0.33)
128	834.56(0.21)	383.39(0.50)	191.63(0.35)	96.48(0.20)	48.22(0.80)

Table C.0.14: Parallel CLONALG: Speedup when Varying the Number of Input Vectors

np→	1	2	4	8	16
M↓	S	S	S	S	S
8	1.00(0.00)	1.96(0.02)	3.81(0.06)	6.85(0.11)	
16	1.00(0.00)	2.21(0.01)	4.35(0.03)	8.70(0.09)	17.06(0.30)
32	1.00(0.00)	1.96(0.00)	3.89(0.01)	7.74(0.11)	15.53(0.12)
64	1.00(0.00)	1.97(0.00)	3.93(0.01)	7.82(0.04)	15.52(0.20)
128	1.00(0.00)	2.18(0.00)	4.36(0.01)	8.65(0.02)	17.31(0.27)

Table C.0.15: Parallel CLONALG: Parallel Efficiency when Varying the Number of Input Vectors

np→	1	2	4	8	16
M↓	E_p	E_p	E_p	E_p	E_p
8	1.00(0.00)	0.98(0.01)	0.95(0.01)	0.86(0.01)	
16	1.00(0.00)	1.10(0.00)	1.09(0.01)	1.09(0.01)	1.07(0.02)
32	1.00(0.00)	0.98(0.00)	0.97(0.00)	0.97(0.01)	0.97(0.01)
64	1.00(0.00)	0.98(0.00)	0.98(0.00)	0.98(0.01)	0.97(0.01)
128	1.00(0.00)	1.09(0.00)	1.09(0.00)	1.08(0.00)	1.08(0.02)

Table C.0.16: Parallel CLONALG: Runtimes when Varying the Length of the Input vector

np→	1	2	4	8	16
L↓	time (s)	time (s)	time (s)	time (s)	time (s)
1	13.76(0.01)	7.11(0.35)	3.70(0.24)	2.01(0.41)	1.01(0.03)
2	14.97(0.02)	7.63(0.09)	3.93(0.03)	2.04(0.04)	1.11(0.10)
4	15.52(0.01)	7.90(0.09)	4.09(0.07)	2.12(0.04)	1.13(0.03)
8	15.49(0.03)	7.86(0.08)	4.06(0.04)	2.14(0.16)	1.13(0.05)
16	32.82(0.02)	16.64(0.07)	8.48(0.12)	4.31(0.04)	2.23(0.08)
32	89.22(0.05)	36.50(0.10)	18.36(0.06)	9.26(0.04)	4.69(0.05)
64	187.88(0.07)	95.97(0.16)	48.35(0.15)	24.24(0.20)	12.14(0.14)
128	607.88(0.31)	316.62(2.14)	162.31(3.49)	79.83(0.99)	40.14(0.81)

Table C.0.17: Parallel CLONALG: Speedup when Varying the Length of the Input Vector

np→	1	2	4	8	16
L↓	S	S	S	S	S
1	1.00(0.00)	1.94(0.08)	3.73(0.18)	6.99(0.83)	13.68(0.43)
2	1.00(0.00)	1.96(0.02)	3.81(0.03)	7.33(0.15)	13.59(0.91)
4	1.00(0.00)	1.97(0.02)	3.80(0.06)	7.32(0.14)	13.79(0.40)
8	1.00(0.00)	1.97(0.02)	3.81(0.03)	7.27(0.41)	13.69(0.55)
16	1.00(0.00)	1.97(0.01)	3.87(0.05)	7.62(0.08)	14.73(0.47)
32	1.00(0.00)	2.44(0.01)	4.86(0.02)	9.64(0.04)	19.01(0.18)
64	1.00(0.00)	1.96(0.00)	3.89(0.01)	7.75(0.06)	15.48(0.17)
128	1.00(0.00)	1.92(0.01)	3.75(0.08)	7.62(0.09)	15.15(0.31)

Table C.0.18: Parallel CLONALG: Parallel Efficiency when Varying the Length of the Input Vector

np→	1	2	4	8	16
L↓	E_p	E_p	E_p	E_p	E_p
1	1.00(0.00)	0.97(0.04)	0.93(0.05)	0.87(0.10)	0.86(0.03)
2	1.00(0.00)	0.98(0.01)	0.95(0.01)	0.92(0.02)	0.85(0.06)
4	1.00(0.00)	0.98(0.01)	0.95(0.01)	0.91(0.02)	0.86(0.02)
8	1.00(0.00)	0.99(0.01)	0.95(0.01)	0.91(0.05)	0.86(0.03)
16	1.00(0.00)	0.99(0.00)	0.97(0.01)	0.95(0.01)	0.92(0.03)
32	1.00(0.00)	1.22(0.00)	1.21(0.00)	1.20(0.01)	1.19(0.01)
64	1.00(0.00)	0.98(0.00)	0.97(0.00)	0.97(0.01)	0.97(0.01)
128	1.00(0.00)	0.96(0.01)	0.94(0.02)	0.95(0.01)	0.95(0.02)

Appendix D

Parallel AIRS Results

This appendix presents results for experiments with the parallel version of AIRS.

Table D.0.1: Iris Results: Concatenation

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	95.69%(3.05)	63.09(3.75)	0.34(0.05)
2	95.64%(3.21)	73.52(4.35)	0.23(0.03)
4	95.00%(3.76)	83.69(3.91)	0.24(0.08)
8	95.82%(3.19)	95.77(3.78)	0.23(0.09)
16	95.38%(2.75)	104.53(2.82)	0.23(0.15)

Table D.0.2: Pima Diabetes Results: Concatenation

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	72.80%(5.04)	279.82(11.05)	3.43(0.12)
2	73.26%(4.81)	317.86(10.46)	2.75(0.05)
4	74.05%(4.91)	356.64(10.42)	2.23(0.06)
8	74.02%(4.68)	400.63(11.90)	2.00(0.07)
16	74.12%(4.87)	445.64(11.19)	1.96(0.16)

Table D.0.3: Sonar Results: Concatenation

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	84.46%(7.68)	173.10(3.73)	54.74(3.03)
2	83.94%(7.88)	179.71(3.09)	32.73(1.80)
4	84.94%(7.88)	184.82(2.54)	19.41(1.25)
8	83.93%(8.48)	187.74(1.87)	11.54(0.93)
16	84.12%(7.86)	189.90(1.39)	6.99(0.54)

Table D.0.4: Concatenation: Speedup and Efficiency

	Iris		Pima Diabetes		Sonar	
np	S	E_p	S	E_p	S	E_p
1	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
2	1.50(0.24)	0.75(0.12)	1.25(0.05)	0.62(0.03)	1.68(0.13)	0.84(0.07)
4	1.51(0.42)	0.38(0.10)	1.54(0.07)	0.38(0.02)	2.83(0.24)	0.71(0.06)
8	1.68(0.55)	0.21(0.07)	1.71(0.08)	0.21(0.01)	4.77(0.46)	0.60(0.06)
16	1.69(0.50)	0.11(0.03)	1.76(0.10)	0.11(0.01)	7.87(0.73)	0.49(0.05)

Table D.0.5: Iris Results: Affinity-Based Merging

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	95.07%(3.18)	61.75(4.26)	0.35(0.02)
2	94.71%(3.44)	67.44(4.33)	0.24(0.04)
4	94.58%(3.85)	72.18(4.30)	0.24(0.06)
8	95.24%(3.67)	79.81(4.20)	0.22(0.06)
16	94.67%(3.04)	84.14(3.97)	0.20(0.08)

Table D.0.6: Pima Diabetes Results: Affinity-Based Merging

NP	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	72.81%(4.62)	276.32(10.56)	3.60(0.06)
2	73.55%(4.54)	307.36(10.66)	2.99(0.06)
4	73.76%(4.40)	339.63(10.90)	2.53(0.08)
8	73.70%(4.49)	372.85(11.05)	2.36(0.11)
16	74.1%3(4.73)	408.83(11.65)	2.36(0.09)

Table D.0.7: Sonar Results: Affinity-Based Merging

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	84.58%(7.90)	173.37(3.79)	54.80(2.77)
2	84.50%(8.36)	179.73(3.26)	32.92(1.94)
4	83.97%(8.25)	184.85(2.29)	19.67(1.25)
8	83.99%(8.17)	187.87(1.86)	11.71(0.84)
16	84.38%(8.15)	189.87(1.33)	7.21(0.59)

Table D.0.8: Affinity-Based Merging: Speedup and Efficiency

	Iris		Pima Diabetes		Sonar	
np	S	E_p	S	E_p	S	E_p
1	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
2	1.47(0.15)	0.74(0.07)	1.20(0.03)	0.60(0.02)	1.67(0.13)	0.84(0.06)
4	1.51(0.34)	0.38(0.08)	1.42(0.05)	0.36(0.01)	2.80(0.23)	0.70(0.06)
8	1.73(0.48)	0.22(0.06)	1.53(0.05)	0.19(0.01)	4.70(0.41)	0.59(0.05)
16	1.86(0.52)	0.12(0.03)	1.53(0.05)	0.10(0.00)	7.65(0.73)	0.48(0.05)

Table D.0.9: Iris Results: Processor Dependent, Affinity-Based Merging

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	95.07%(3.18)	61.75(4.26)	0.35(0.02)
2	95.13%(3.78)	58.84(4.09)	0.23(0.02)
4	94.49%(3.94)	53.12(3.85)	0.23(0.05)
8	94.82%(4.22)	50.97(4.37)	0.23(0.21)
16	94.36%(3.58)	46.84(3.28)	0.21(0.07)

Table D.0.10: Pima Diabetes Results: Processor Dependent, Affinity-Based Merging

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	72.81%(4.62)	276.32(10.56)	3.60(0.06)
2	73.39%(4.72)	259.44(9.89)	2.83(0.06)
4	73.45%(4.33)	228.88(9.22)	2.17(0.06)
8	73.64%(4.87)	192.10(8.61)	1.78(0.04)
16	73.40%(5.19)	164.75(7.93)	1.57(0.12)

Table D.0.11: Sonar Results: Processor Dependent, Affinity-Based Merging

np	Test Set Accuracy	Memory Cells	Overall Runtime(s)
1	84.58%(7.90)	173.37(3.79)	54.80(2.77)
2	84.02%(8.68)	173.65(3.59)	32.69(1.91)
4	83.96%(8.26)	175.75(3.09)	19.59(1.28)
8	83.72%(8.50)	177.21(3.33)	11.61(0.85)
16	83.91%(8.01)	179.65(3.04)	7.11(0.59)

Table D.0.12: Iris Results: Varying the "Dampener": Accuracy

np	1	2	4	8	16
D	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0.1	95.07%(3.18)	95.02%(3.64)	94.47%(3.82)	95.36%(3.92)	94.62%(3.43)
0.2	95.07%(3.18)	95.13%(3.78)	94.49%(3.94)	94.82%(4.22)	94.36%(3.58)
0.4	95.07%(3.18)	95.18%(3.22)	93.73%(4.38)	95.29%(4.11)	94.22%(3.95)
0.6	95.07%(3.18)	95.16%(3.27)	94.20%(4.33)	94.89%(4.28)	94.38%(4.05)
0.8	95.07%(3.18)	94.62%(4.08)	93.38%(4.95)	94.80%(4.04)	94.04%(3.61)
1	95.07%(3.18)	93.98%(4.60)	93.27%(5.00)	93.87%(4.21)	93.18%(4.67)

Table D.0.13: Iris Results: Varying the "Dampener": Memory Cells

np	1	2	4	8	16
D	MCs	MCs	MCs	MCs	MCs
0.1	61.75(4.26)	62.57(4.23)	62.26(3.80)	64.39(4.13)	63.93(3.50)
0.2	61.75(4.26)	58.84(4.09)	53.12(3.85)	50.97(4.37)	46.84(3.28)
0.4	61.75(4.26)	50.69(3.84)	39.31(3.42)	32.54(3.28)	26.39(2.71)
0.6	61.75(4.26)	43.74(3.34)	28.93(3.06)	21.96(2.28)	16.71(1.77)
0.8	61.75(4.26)	37.55(3.25)	22.24(2.45)	15.62(2.09)	11.32(1.45)
1	61.75(4.26)	32.82(2.78)	17.59(2.16)	11.65(1.44)	9.04(1.09)

Table D.0.14: Iris Results: Varying the "Dampener": Run Time

np	1	2	4	8	16
D	time (s)	time (s)	time (s)	time (s)	time (s)
0.1	0.35(0.02)	0.23(0.09)	0.25(0.06)	0.24(0.06)	0.23(0.09)
0.2	0.35(0.02)	0.23(0.02)	0.23(0.05)	0.23(0.21)	0.21(0.07)
0.4	0.35(0.02)	0.23(0.03)	0.22(0.04)	0.21(0.11)	0.20(0.07)
0.6	0.35(0.02)	0.23(0.02)	0.22(0.05)	0.20(0.07)	0.20(0.06)
0.8	0.35(0.02)	0.22(0.03)	0.22(0.05)	0.21(0.09)	0.21(0.06)
1	0.35(0.02)	0.22(0.03)	0.21(0.05)	0.20(0.07)	0.20(0.07)

Table D.0.15: Pima Diabetes Results: Varying the "Dampener": Accuracy

np	1	2	4	8	16
D	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0.1	72.81%(4.62)	73.56%(4.59)	73.85%(4.43)	73.85%(4.97)	73.69%(4.71)
0.2	72.81%(4.62)	73.39%(4.72)	73.45%(4.33)	73.64%(4.87)	73.40%(5.19)
0.4	72.81%(4.62)	73.25%(4.38)	73.25%(4.84)	73.15%(4.87)	72.44%(5.11)
0.6	72.81%(4.62)	73.30%(4.54)	72.67%(4.91)	70.90%(5.35)	69.43%(6.25)
0.8	72.81%(4.62)	72.91%(4.67)	71.82%(4.87)	69.49%(5.58)	63.68%(9.33)
1	72.81%(4.62)	72.66%(4.93)	70.21%(5.48)	65.54%(7.95)	55.34%(14.72)

Table D.0.16: Pima Diabetes Results: Varying the "Dampener": Memory Cells

np	1	2	4	8	16
D	MCs	MCs	MCs	MCs	MCs
0.1	276.32(10.56)	282.58(10.29)	281.62(10.11)	273.44(9.12)	265.97(9.70)
0.2	276.32(10.56)	259.44(9.89)	228.88(9.22)	192.10(8.61)	164.75(7.93)
0.4	276.32(10.56)	213.34(8.19)	144.61(6.92)	93.42(5.72)	64.66(5.37)
0.6	276.32(10.56)	173.13(7.69)	90.69(5.78)	48.09(4.11)	29.89(3.19)
0.8	276.32(10.56)	138.94(6.04)	59.00(4.33)	27.43(2.99)	16.21(2.40)
1	276.32(10.56)	112.04(5.51)	39.21(3.37)	17.15(2.36)	9.70(1.72)

Table D.0.17: Pima Diabetes Results: Varying the "Dampener": Run Time

np	1	2	4	8	16
D	time (s)	time (s)	time (s)	time (s)	time (s)
0.1	3.60(0.06)	2.91(0.07)	2.35(0.07)	2.04(0.04)	1.90(0.12)
0.2	3.60(0.06)	2.83(0.06)	2.17(0.06)	1.78(0.04)	1.57(0.12)
0.4	3.60(0.06)	2.67(0.06)	1.91(0.07)	1.49(0.04)	1.30(0.05)
0.6	3.60(0.06)	2.53(0.05)	1.75(0.10)	1.37(0.03)	1.22(0.04)
0.8	3.60(0.06)	2.41(0.05)	1.66(0.11)	1.33(0.06)	1.20(0.05)
1	3.60(0.06)	2.33(0.05)	1.60(0.05)	1.31(0.06)	1.19(0.11)

Table D.0.18: Sonar Results: Varying the "Dampener": Accuracy

np	1	2	4	8	16
D	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
0.1	84.58%(7.90)	84.38%(8.43)	84.12%(8.27)	84.02%(8.66)	83.67%(8.12)
0.2	84.58%(7.90)	84.02%(8.68)	83.96%(8.26)	83.72%(8.50)	83.91%(8.01)
0.4	84.58%(7.90)	85.05%(8.11)	84.33%(8.57)	83.33%(8.79)	84.31%(8.17)
0.6	84.58%(7.90)	84.63%(8.64)	84.10%(8.45)	83.96%(8.37)	83.65%(8.08)
0.8	84.58%(7.90)	84.54%(7.76)	83.91%(8.66)	83.64%(8.38)	83.46%(9.09)
1	84.58%(7.90)	84.26%(7.89)	84.38%(8.15)	83.19%(8.28)	82.16%(9.37)

Table D.0.19: Sonar Results: Varying the "Dampener": Memory Cells

np	1	2	4	8	16
D	MCs	MCs	MCs	MCs	MCs
0.1	173.37(3.79)	176.55(3.13)	180.07(2.96)	182.80(2.43)	185.14(2.31)
0.2	173.37(3.79)	173.65(3.59)	175.75(3.09)	177.21(3.33)	179.65(3.04)
0.4	173.37(3.79)	168.01(3.84)	166.27(3.87)	166.85(3.72)	167.11(4.26)
0.6	173.37(3.79)	162.58(4.02)	158.27(4.36)	154.97(4.63)	151.37(4.61)
0.8	173.37(3.79)	157.52(4.51)	150.40(4.42)	140.96(4.79)	128.69(4.69)
1	173.37(3.79)	152.27(4.34)	140.84(4.87)	123.91(4.96)	101.62(4.78)

Table D.0.20: Sonar Results: Varying the "Dampener": Run Time

np	1	2	4	8	16
D	time (s)	time (s)	time (s)	time (s)	time (s)
0.1	54.80(2.77)	32.73(1.82)	21.14(8.36)	11.76(0.96)	7.18(0.59)
0.2	54.80(2.77)	32.69(1.91)	19.59(1.28)	11.61(0.85)	7.11(0.59)
0.4	54.80(2.77)	32.91(1.89)	19.55(1.30)	11.61(0.82)	7.14(0.58)
0.6	54.80(2.77)	32.93(1.86)	19.61(1.34)	11.56(0.83)	7.01(0.56)
0.8	54.80(2.77)	32.93(1.98)	19.63(1.35)	11.54(0.90)	6.90(0.56)
1	54.80(2.77)	32.55(1.98)	19.47(1.30)	11.45(0.91)	6.82(0.58)

Table D.0.21: Procssor Dependent, Affinity-Based Merging: Speedup and Efficiency

	Iris		Pima Diabetes		Sonar	
np	S	E_p	S	E_p	S	E_p
1	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)	1.00(0.00)
2	1.51(0.12)	0.76(0.06)	1.27(0.03)	0.64(0.02)	1.68(0.13)	0.84(0.06)
4	1.59(0.37)	0.40(0.09)	1.66(0.05)	0.41(0.01)	2.81(0.23)	0.70(0.06)
8	1.79(0.55)	0.22(0.07)	2.02(0.05)	0.25(0.01)	4.74(0.42)	0.59(0.05)
16	1.83(0.56)	0.11(0.03)	2.29(0.09)	0.14(0.01)	7.75(0.74)	0.48(0.05)

```
;*********************************************************
; Parameter file for DGP/2
num_features = 64 ; Number of features
max_feature_value = 20 ; Maximum feature value
num_peaks = 5 ; Number of peaks in the final space
num_instances = 320 ; Number of instances generated
proto_seed = 2398 ; Initial random seed
range = 3 ; Range for positive class membership
percentage = 67 ; Percentage of positive instances
trunc_flag = 0 ; Out of range instance disposition flag
out_file_name = 320.64.dat ; Filename for instances
stat_file_name = 320.64.sts ; Filename for run statistics
; End of parameter file
```

Figure D.0.1: Sample DGP-2 Parameter File

Table D.0.22: Parallel AIRS: Run Times when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	time(s)	time(s)	time(s)	time(s)	time(s)
32	0.30(0.04)	0.21(0.04)	0.23(0.07)	0.20(0.08)	0.20(0.12)
64	0.64(0.06)	0.44(0.05)	0.36(0.06)	0.31(0.06)	0.28(0.08)
128	1.45(0.07)	1.05(0.06)	0.78(0.05)	0.65(0.07)	0.57(0.07)
256	3.64(0.12)	2.89(0.11)	2.17(0.07)	1.86(0.06)	1.67(0.05)
512	10.01(0.23)	9.01(0.21)	7.16(0.16)	6.25(0.38)	5.78(0.10)

Table D.0.23: Parallel AIRS: Test Set Accuracy when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
32	57.17%(21.05)	54.00%(13.53)	52.58%(11.15)	72.00%(9.89)	69.42%(15.36)
64	51.38%(8.86)	50.83%(7.16)	47.96%(13.36)	52.96%(16.84)	55.67%(9.08)
128	59.88%(6.84)	56.27%(8.18)	54.44%(6.72)	54.15%(7.74)	58.10%(5.22)
256	63.07%(5.45)	59.35%(5.16)	52.84%(5.11)	53.28%(6.63)	55.20%(6.12)
512	63.52%(5.11)	65.04%(5.19)	62.48%(4.99)	59.60%(5.69)	58.74%(4.81)

Table D.0.24: Parallel AIRS: Number of Memory Cells when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	MCs	MCs	MCs	MCs	MCs
32	23.06(2.31)	23.73(2.13)	25.32(1.74)	27.26(1.52)	29.68(1.24)
64	45.47(3.15)	45.03(3.07)	45.65(2.91)	47.63(2.77)	52.05(2.24)
128	89.07(4.97)	87.26(4.61)	86.83(4.33)	86.38(4.24)	88.71(3.87)
256	172.65(7.15)	169.09(6.02)	166.84(5.94)	166.13(5.63)	163.53(5.31)
512	337.24(10.72)	334.36(9.86)	327.41(8.51)	318.05(7.63)	313.77(7.60)

Table D.0.25: Parallel AIRS: Performance Metrics when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	S	S	S	S	S
32	1.00(0.00)	1.48(0.28)	1.43(0.43)	1.65(0.58)	1.74(0.62)
64	1.00(0.00)	1.48(0.20)	1.81(0.29)	2.15(0.42)	2.39(0.55)
128	1.00(0.00)	1.38(0.10)	1.88(0.15)	2.26(0.21)	2.57(0.26)
256	1.00(0.00)	1.26(0.06)	1.68(0.07)	1.96(0.09)	2.18(0.09)
512	1.00(0.00)	1.11(0.04)	1.40(0.04)	1.61(0.07)	1.73(0.05)
N↓	E_p	E_p	E_p	E_p	E_p
32	1.00(0.00)	0.74(0.14)	0.36(0.11)	0.21(0.07)	0.11(0.04)
64	1.00(0.00)	0.74(0.10)	0.45(0.07)	0.27(0.05)	0.15(0.03)
128	1.00(0.00)	0.69(0.05)	0.47(0.04)	0.28(0.03)	0.16(0.02)
256	1.00(0.00)	0.63(0.03)	0.42(0.02)	0.24(0.01)	0.14(0.01)
512	1.00(0.00)	0.56(0.02)	0.35(0.01)	0.20(0.01)	0.11(0.00)

Table D.0.26: Parallel AIRS: Run Times when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	time(s)	time(s)	time(s)	time(s)	time(s)
1	0.83(0.04)	0.49(0.03)	0.36(0.05)	0.31(0.18)	0.27(0.08)
2	0.79(0.04)	0.52(0.04)	0.41(0.05)	0.35(0.06)	0.31(0.06)
4	1.00(0.04)	0.74(0.03)	0.59(0.07)	0.54(0.15)	0.48(0.11)
8	1.28(0.05)	0.97(0.03)	0.77(0.04)	0.67(0.03)	0.62(0.03)
16	1.68(0.06)	1.29(0.04)	0.99(0.04)	0.84(0.03)	0.76(0.03)
32	2.39(0.07)	1.87(0.07)	1.41(0.05)	1.19(0.05)	1.07(0.03)
64	3.64(0.12)	2.89(0.11)	2.17(0.07)	1.86(0.06)	1.67(0.05)
128	5.83(0.19)	4.76(0.17)	3.60(0.11)	3.05(0.08)	2.80(0.07)

Table D.0.27: Parallel AIRS: Performance Metrics when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	S	S	S	S	S
1	1.00(0.00)	1.70(0.11)	2.32(0.28)	2.88(0.56)	3.22(0.67)
2	1.00(0.00)	1.54(0.11)	1.98(0.21)	2.29(0.32)	2.64(0.39)
4	1.00(0.00)	1.36(0.08)	1.70(0.16)	1.91(0.23)	2.14(0.22)
8	1.00(0.00)	1.32(0.06)	1.67(0.10)	1.91(0.11)	2.08(0.12)
16	1.00(0.00)	1.31(0.06)	1.70(0.09)	2.00(0.10)	2.21(0.11)
32	1.00(0.00)	1.28(0.06)	1.70(0.07)	2.02(0.10)	2.23(0.10)
64	1.00(0.00)	1.26(0.06)	1.68(0.07)	1.96(0.09)	2.18(0.09)
128	1.00(0.00)	1.23(0.06)	1.62(0.07)	1.91(0.08)	2.08(0.08)
L↓	E_p	E_p	E_p	E_p	E_p
1	1.00(0.00)	0.85(0.05)	0.58(0.07)	0.36(0.07)	0.20(0.04)
2	1.00(0.00)	0.77(0.06)	0.49(0.05)	0.29(0.04)	0.16(0.02)
4	1.00(0.00)	0.68(0.04)	0.42(0.04)	0.24(0.03)	0.13(0.01)
8	1.00(0.00)	0.66(0.03)	0.42(0.03)	0.24(0.01)	0.13(0.01)
16	1.00(0.00)	0.65(0.03)	0.43(0.02)	0.25(0.01)	0.14(0.01)
32	1.00(0.00)	0.64(0.03)	0.42(0.02)	0.25(0.01)	0.14(0.01)
64	1.00(0.00)	0.63(0.03)	0.42(0.02)	0.24(0.01)	0.14(0.01)
128	1.00(0.00)	0.61(0.03)	0.40(0.02)	0.24(0.01)	0.13(0.01)

Table D.0.28: Parallel AIRS: Test Set Accuracy when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
1	97.47%(2.95)	96.72%(3.83)	94.78%(4.13)	88.33%(7.51)	88.30%(8.31)
2	92.05%(3.22)	91.17%(3.82)	89.78%(3.96)	88.91%(3.99)	89.10%(3.94)
4	77.82%(7.24)	76.67%(7.13)	75.41%(6.42)	73.88%(6.17)	77.71%(6.34)
8	71.30%(5.29)	70.97%(4.20)	70.56%(4.65)	70.60%(4.59)	73.77%(4.26)
16	63.66%(6.22)	64.83%(5.74)	64.78%(6.12)	61.75%(5.57)	62.44%(5.21)
32	63.16%(6.13)	61.02%(5.48)	59.11%(6.41)	58.78%(5.24)	58.58%(5.31)
64	63.07%(5.45)	59.35%(5.16)	52.84%(5.11)	53.28%(6.63)	55.20%(6.12)
128	62.93%(4.47)	59.79%(4.45)	55.07%(4.95)	54.32%(5.33)	54.41%(6.88)

Table D.0.29: Parallel AIRS: Number of Memory Cells when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	MCs	MCs	MCs	MCs	MCs
1	20.87(1.64)	22.83(2.25)	25.72(2.08)	26.29(2.56)	27.11(2.17)
2	94.59(4.83)	99.17(4.74)	101.22(4.60)	99.07(4.07)	96.73(3.60)
4	198.45(5.31)	204.65(5.74)	205.95(5.30)	207.64(5.18)	206.10(4.98)
8	225.91(4.75)	227.02(4.33)	226.55(4.58)	225.77(4.13)	227.41(4.70)
16	222.24(5.41)	218.00(5.57)	214.17(5.14)	212.11(4.99)	209.85(5.03)
32	198.43(6.09)	194.73(7.34)	190.47(5.35)	186.15(5.78)	185.60(5.67)
64	172.65(7.15)	169.09(6.02)	166.84(5.94)	166.13(5.63)	163.53(5.31)
128	148.37(7.89)	147.90(7.30)	144.85(5.97)	143.57(5.34)	144.07(4.74)

Appendix E

Distributed AIRS Results

This appendix presents results for experiments with the distributed version of AIRS.

Table E.0.1: Distributed Iris: Training Set Accuracy

np	Global Acc.	Min. Acc.	Max. Acc.
1	97.94%(0.99)	97.94%(0.99)	97.94%(0.99)
2	98.47%(0.96)	97.59%(1.36)	99.36%(0.96)
4	97.94%(1.20)	95.00%(2.72)	99.91%(0.54)
8	97.74%(1.03)	92.71%(2.60)	100.00%(0.00)
16	97.07%(1.00)	80.52%(6.99)	100.00%(0.00)

Table E.0.2: Distributed Pima Diabetes: Training Set Accuracy

np	Global Acc.	Min. Acc.	Max. Acc.
1	76.47%(1.25)	76.47%(1.25)	76.47%(1.25)
2	76.17%(1.21)	74.79%(1.63)	77.55%(1.44)
4	76.01%(1.12)	73.41%(1.70)	78.65%(1.69)
8	75.84%(1.30)	69.90%(2.67)	82.00%(2.67)
16	75.22%(1.40)	63.07%(4.22)	85.68%(2.52)

Table E.0.3: Distributed Sonar: Training Set Accuracy

np	Global Acc.	Min. Acc.	Max. Acc.
1	97.79%(1.12)	97.79%(1.12)	97.79%(1.12)
2	96.72%(1.29)	95.67%(1.73)	97.77%(1.32)
4	95.06%(1.50)	91.49%(2.74)	98.13%(1.59)
8	93.68%(1.69)	86.05%(4.10)	99.37%(1.49)
16	93.00%(1.74)	77.91%(6.25)	100.00%(0.00)

Table E.0.4: Distributed AIRS: Run Times when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	time(s)	time(s)	time(s)	time(s)	time(s)
32	0.29(0.06)	0.16(0.04)	0.17(0.06)	0.17(0.10)	0.21(0.09)
64	0.59(0.06)	0.30(0.04)	0.23(0.06)	0.19(0.08)	0.22(0.08)
128	1.37(0.10)	0.63(0.05)	0.36(0.06)	0.24(0.07)	0.24(0.08)
256	3.34(0.11)	1.41(0.09)	0.67(0.05)	0.36(0.07)	0.26(0.09)
512	9.11(0.20)	3.44(0.09)	1.45(0.07)	0.69(0.05)	0.41(0.17)

Table E.0.5: Distributed AIRS: Performance Metrics when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	S	S	S	S	S
32	1.00(0.00)	1.87(0.49)	1.91(0.78)	2.07(1.04)	1.51(0.53)
64	1.00(0.00)	1.96(0.29)	2.68(0.67)	3.40(1.17)	2.89(0.83)
128	1.00(0.00)	2.19(0.22)	3.90(0.60)	6.16(1.37)	6.20(1.36)
256	1.00(0.00)	2.38(0.15)	5.05(0.39)	9.47(1.28)	13.48(2.37)
512	1.00(0.00)	2.65(0.09)	6.27(0.29)	13.24(0.87)	22.97(2.85)
N↓	E_p	E_p	E_p	E_p	E_p
32	1.00(0.00)	0.94(0.24)	0.48(0.19)	0.26(0.13)	0.09(0.03)
64	1.00(0.00)	0.98(0.14)	0.67(0.17)	0.42(0.15)	0.18(0.05)
128	1.00(0.00)	1.09(0.11)	0.98(0.15)	0.77(0.17)	0.39(0.09)
256	1.00(0.00)	1.19(0.07)	1.26(0.10)	1.18(0.16)	0.84(0.15)
512	1.00(0.00)	1.33(0.05)	1.57(0.07)	1.66(0.11)	1.44(0.18)

Table E.0.6: Distributed AIRS: Run Times when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	time(s)	time(s)	time(s)	time(s)	time(s)
1	0.83(0.03)	0.41(0.04)	0.29(0.06)	0.24(0.09)	0.25(0.19)
2	0.76(0.05)	0.37(0.03)	0.27(0.07)	0.21(0.07)	0.22(0.07)
4	0.85(0.03)	0.40(0.03)	0.27(0.06)	0.22(0.07)	0.27(0.09)
8	1.08(0.06)	0.48(0.05)	0.30(0.06)	0.21(0.06)	0.25(0.08)
16	1.46(0.07)	0.63(0.04)	0.36(0.07)	0.24(0.08)	0.25(0.11)
32	2.17(0.11)	0.93(0.06)	0.47(0.08)	0.28(0.06)	0.27(0.10)
64	3.34(0.11)	1.41(0.09)	0.67(0.05)	0.36(0.07)	0.26(0.09)
128	5.44(0.17)	2.27(0.11)	1.05(0.08)	0.55(0.11)	0.33(0.05)

Table E.0.7: Distributed AIRS: Performance Metrics when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	S	S	S	S	S
1	1.00(0.00)	2.03(0.15)	3.00(0.57)	3.95(1.28)	3.93(1.29)
2	1.00(0.00)	2.07(0.17)	2.97(0.60)	3.90(1.16)	3.77(1.02)
4	1.00(0.00)	2.15(0.14)	3.32(0.65)	4.18(1.28)	3.49(1.00)
8	1.00(0.00)	2.24(0.18)	3.74(0.72)	5.52(1.47)	4.78(1.27)
16	1.00(0.00)	2.32(0.16)	4.18(0.64)	6.49(1.67)	6.49(1.77)
32	1.00(0.00)	2.35(0.18)	4.68(0.58)	7.95(1.40)	8.74(2.16)
64	1.00(0.00)	2.38(0.15)	5.05(0.39)	9.47(1.28)	13.48(2.37)
128	1.00(0.00)	2.40(0.13)	5.20(0.38)	10.12(1.04)	16.56(1.85)
L↓	E_p	E_p	E_p	E_p	E_p
1	1.00(0.00)	1.02(0.08)	0.75(0.14)	0.49(0.16)	0.25(0.08)
2	1.00(0.00)	1.04(0.08)	0.74(0.15)	0.49(0.15)	0.24(0.06)
4	1.00(0.00)	1.07(0.07)	0.83(0.16)	0.52(0.16)	0.22(0.06)
8	1.00(0.00)	1.12(0.09)	0.93(0.18)	0.69(0.18)	0.30(0.08)
16	1.00(0.00)	1.16(0.08)	1.04(0.16)	0.81(0.21)	0.41(0.11)
32	1.00(0.00)	1.17(0.09)	1.17(0.14)	0.99(0.17)	0.55(0.14)
64	1.00(0.00)	1.19(0.07)	1.26(0.10)	1.18(0.16)	0.84(0.15)
128	1.00(0.00)	1.20(0.07)	1.30(0.09)	1.26(0.13)	1.04(0.12)

Table E.0.8: Distributed AIRS: Global Test Set Accuracy when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
32	56.75%(21.32)	62.00%(10.63)	59.50%(13.87)	72.08%(10.08)	51.75%(20.79)
64	52.08%(8.86)	55.67%(9.67)	58.08%(17.28)	61.46%(13.18)	56.58%(12.60)
128	59.06%(7.44)	59.23%(7.34)	58.63%(8.27)	55.88%(7.98)	57.90%(8.63)
256	62.52%(4.89)	59.02%(5.63)	54.56%(5.96)	57.50%(4.10)	57.40%(5.09)
512	63.88%(5.23)	62.88%(5.42)	61.48%(6.52)	59.02%(5.81)	55.43%(5.24)

Table E.0.9: Distributed AIRS: Global Number of Memory Cells when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	MCs	MCs	MCs	MCs	MCs
32	22.98(2.68)	25.83(2.20)	27.94(1.56)	30.24(1.20)	31.63(0.57)
64	45.53(3.39)	49.64(3.40)	53.27(2.62)	56.89(2.55)	60.73(1.58)
128	88.91(4.82)	95.39(4.64)	102.18(4.73)	107.99(4.10)	114.00(3.45)
256	172.15(7.70)	185.25(7.13)	196.83(6.52)	210.01(5.82)	217.89(5.68)
512	338.06(11.30)	362.17(9.45)	385.57(9.24)	408.22(8.87)	424.48(8.34)

Table E.0.10: Distributed AIRS: Local Minimum Test Set Accuracy when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Min. Acc.	Min. Acc.	Min. Acc.	Min. Acc.	Min. Acc.
32	56.75%(21.32)	41.17%(13.93)	30.33%(24.51)	0.00%(0.00)	0.00%(0.00)
64	52.08%(8.86)	46.58%(12.86)	42.83%(19.72)	21.67%(24.86)	0.00%(0.00)
128	59.06%(7.44)	51.46%(9.23)	41.58%(14.45)	21.17%(14.98)	0.00%(0.00)
256	62.52%(4.89)	55.23%(6.55)	41.17%(8.17)	35.25%(8.20)	18.83%(11.92)
512	63.88%(5.23)	58.58%(6.86)	51.58%(7.94)	40.42%(8.80)	25.33%(8.44)

Table E.0.11: Distributed AIRS: Local Maximum Test Set Accuracy when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Max. Acc.	Max. Acc.	Max. Acc.	Max. Acc.	Max. Acc.
32	56.75%(21.32)	82.83%(17.89)	98.00%(9.83)	100.00%(0.00)	100.00%(0.00)
64	52.08%(8.86)	64.75%(9.94)	73.67%(17.08)	100.00%(0.00)	100.00%(0.00)
128	59.06%(7.44)	67.00%(9.77)	77.58%(10.77)	93.00%(11.26)	100.00%(0.00)
256	62.52%(4.89)	62.81%(6.09)	67.54%(7.99)	80.33%(9.42)	93.50%(11.00)
512	63.88%(5.23)	67.17%(5.72)	70.63%(7.82)	79.13%(9.15)	83.50%(9.08)

Table E.0.12: Distributed AIRS: Local Minimum Number of Memory Cells when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Min MCs	Min MCs	Min MCs	Min MCs	Min MCs
32	22.98(2.68)	12.07(1.38)	6.07(0.71)	3.09	1.67(0.47)
64	45.53(3.39)	23.45(2.11)	11.80(1.14)	5.83	2.83(0.42)
128	88.91(4.82)	46.05(2.65)	23.17(1.86)	11.36	5.39(0.64)
256	172.15(7.70)	89.75(4.01)	45.65(2.67)	23.01	11.03(0.98)
512	338.06(11.30)	177.17(5.38)	91.35(3.38)	46.39	22.66(1.23)

Table E.0.13: Distributed AIRS: Local Maximum Number of Memory Cells when Varying the Number of Training Vectors

np→	1	2	4	8	16
N↓	Max MCs	Max MCs	Max MCs	Max MCs	Max MCs
32	22.98(2.68)	13.76(1.27)	7.83(0.37)	4.00(0.00)	2.00(0.00)
64	45.53(3.39)	26.19(1.97)	14.79(0.81)	7.97(0.16)	4.00(0.00)
128	88.91(4.82)	49.34(2.65)	27.85(1.44)	15.31(0.63)	8.00(0.00)
256	172.15(7.70)	95.50(4.19)	52.88(2.16)	29.15(1.17)	15.68(0.47)
512	338.06(11.30)	185.00(6.00)	101.35(3.44)	55.45(1.77)	30.09(0.92)

Table E.0.14: Distributed AIRS: Global Test Set Accuracy when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	Accuracy	Accuracy	Accuracy	Accuracy	Accuracy
1	97.51%(2.45)	97.78%(2.09)	95.51%(2.38)	87.83%(2.96)	82.74%(3.23)
2	92.32%(3.22)	88.24%(3.53)	84.53%(3.77)	82.98%(3.87)	80.34%(3.25)
4	78.11%(6.69)	74.29%(6.83)	71.36%(6.15)	69.28%(5.91)	73.10%(8.34)
8	70.95%(4.88)	70.88%(5.16)	71.73%(5.42)	69.59%(4.80)	70.78%(3.72)
16	64.34%(6.39)	62.46%(6.61)	60.57%(6.77)	62.40%(6.98)	62.82%(5.87)
32	62.69%(5.75)	61.03%(5.82)	60.11%(5.28)	56.24%(6.39)	54.41%(5.44)
64	62.84%(4.72)	59.11%(5.61)	54.19%(5.64)	58.13%(3.87)	58.61%(4.91)
128	63.18%(3.79)	61.21%(5.12)	58.41%(4.85)	56.86%(4.65)	53.67%(6.40)

Table E.0.15: Distributed AIRS: Global Number of Memory Cells when Varying the Lenght of the Input Vectors

np→	1	2	4	8	16
L↓	MCs	MCs	MCs	MCs	MCs
1	21.35(1.84)	37.89(2.44)	64.70(2.64)	93.75(2.25)	133.37(2.42)
2	97.73(4.44)	125.69(5.23)	154.89(5.56)	185.58(4.84)	212.92(5.19)
4	199.28(6.10)	213.25(5.16)	224.87(4.37)	235.37(3.96)	242.66(3.46)
8	226.51(4.85)	232.35(4.30)	236.93(3.76)	241.90(3.78)	246.72(2.89)
16	221.37(5.67)	226.37(5.00)	231.32(4.57)	237.40(3.95)	243.18(3.34)
32	199.03(5.91)	207.42(6.06)	215.66(5.79)	223.67(5.04)	230.97(4.87)
64	172.15(7.70)	185.25(7.13)	196.83(6.52)	210.01(5.82)	217.89(5.68)
128	149.71(7.59)	164.80(7.72)	178.07(6.97)	190.45(7.13)	204.02(5.68)

Table E.0.16: Distributed AIRS: Local Minimum Test Set Accuracy when Varying the Length of the Training Vectors

np→	1	2	4	8	16
L↓	Min. Acc.	Min. Acc.	Min. Acc.	Min. Acc.	Min. Acc.
1	97.51%(2.45)	96.27%(2.98)	88.92%(5.28)	64.92%(10.90)	42.33%(11.57)
2	92.32%(3.22)	85.46%(4.30)	75.42%(7.30)	65.67%(9.07)	46.67%(9.01)
4	78.11%(6.69)	69.29%(8.51)	60.42%(8.86)	43.25%(13.48)	24.83%(20.38)
8	70.95%(4.88)	66.46%(6.44)	59.13%(9.12)	44.83%(10.95)	29.17%(13.71)
16	64.34%(6.39)	57.42%(7.94)	45.17%(12.08)	34.50%(13.94)	19.00%(14.39)
32	62.69%(5.75)	54.13%(6.99)	46.42%(8.24)	31.25%(9.90)	11.67%(12.84)
64	62.84%(4.72)	55.42%(6.76)	41.17%(7.87)	36.00%(7.23)	19.33%(11.27)
128	63.18%(3.79)	56.42%(5.61)	46.63%(7.01)	33.25%(12.27)	13.33%(13.79)

Table E.0.17: Distributed AIRS: Local Maximum Test Set Accuracy when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	Max. Acc.	Max. Acc.	Max. Acc.	Max. Acc.	Max. Acc.
1	97.51%(2.45)	99.29%(1.70)	99.75%(1.23)	100.00%(0.00)	100.00%(0.00)
2	92.32%(3.22)	91.02%(4.00)	92.58%(4.77)	97.33%(5.14)	100.00%(0.00)
4	78.11%(6.69)	79.29%(6.47)	83.42%(7.24)	94.00%(8.28)	100.00%(0.00)
8	70.95%(4.88)	75.29%(6.17)	83.29%(5.54)	92.33%(7.90)	100.00%(0.00)
16	64.34%(6.39)	67.50%(7.16)	74.08%(7.81)	84.33%(7.96)	99.00%(4.92)
32	62.69%(5.75)	67.94%(8.26)	75.38%(8.48)	80.50%(9.66)	94.67%(10.28)
64	62.84%(4.72)	62.81%(6.04)	67.21%(7.27)	80.42%(9.21)	97.50%(7.53)
128	63.18%(3.79)	66.00%(6.47)	68.83%(6.39)	77.33%(5.68)	96.50%(8.70)

Table E.0.18: Distributed AIRS: Local Minimum Number of Memory Cells when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	Min MCs	Min MCs	Min MCs	Min MCs	Min MCs
1	21.35(1.84)	17.60(1.15)	14.63(0.85)	10.09(0.33)	6.94(0.24)
2	97.73(4.44)	60.75(2.92)	35.84(1.97)	20.54(1.19)	11.24(0.66)
4	199.28(6.10)	104.58(3.00)	53.74(1.86)	27.25(1.07)	13.37(0.70)
8	226.51(4.85)	114.55(2.63)	56.93(1.56)	28.40(1.06)	13.83(0.75)
16	221.37(5.67)	110.97(3.15)	55.56(1.83)	27.29(1.16)	13.35(0.72)
32	199.03(5.91)	101.14(3.56)	50.75(2.30)	25.39(1.19)	12.19(0.82)
64	172.15(7.70)	89.75(4.01)	45.65(2.67)	23.01(1.65)	11.03(0.98)
128	149.71(7.59)	78.95(4.68)	40.77(2.58)	20.07(1.68)	9.82(1.02)

Table E.0.19: Distributed AIRS: Local Maximum Number of Memory Cells when Varying the Length of the Input Vectors

np→	1	2	4	8	16
L↓	Max MCs	Max MCs	Max MCs	Max MCs	Max MCs
1	21.35(1.84)	20.29(1.78)	18.43(1.15)	13.75(0.57)	10.01(0.41)
2	97.73(4.44)	64.94(3.04)	41.68(1.98)	25.73(1.07)	15.23(0.51)
4	199.28(6.10)	108.67(3.06)	58.65(1.51)	31.26(0.60)	16.00(0.00)
8	226.51(4.85)	117.79(2.46)	61.23(1.31)	31.73(0.44)	16.00(0.00)
16	221.37(5.67)	115.40(2.79)	60.04(1.27)	31.43(0.62)	16.00(0.00)
32	199.03(5.91)	106.28(3.55)	56.77(1.75)	30.25(0.90)	15.95(0.21)
64	172.15(7.70)	95.50(4.19)	52.88(2.16)	29.15(1.17)	15.68(0.47)
128	149.71(7.59)	85.85(4.47)	48.26(2.41)	27.15(1.24)	15.19(0.60)