DARWINIAN GRADIENT DESCENT

by

Michael Levin

Submitted to the

Faculty of the College of Arts and Sciences

of American University

in Partial Fulfillment of

the Requirements for the Degree

of Master of Science

in

Computer Science

Chair: ______________________________
Professor Michael Black

____
Professor Gregory Bard

____
Professor Angela Wu

____
Dean of the College

____
Date

2010
American University
Washington, D.C. 20016

DARWINIAN GRADIENT DESCENT

by

Michael Levin

Submitted to the

Faculty of the College of Arts and Sciences

of American University

in Partial Fulfillment of

the Requirements for the Degree

of Master of Science

in

Computer Science

Chair: ______________________________
Professor Michael Black

____
Professor Gregory Bard

____
Professor Angela Wu

____
Dean of the College

____
Date

2010
American University
Washington, D.C. 20016

© COPYRIGHT

by

Michael Levin

2010

ALL RIGHTS RESERVED

DARWINIAN GRADIENT DESCENT

by

Michael Levin

ABSTRACT

Large optimization problems of many variables can be difficult to solve and very computationally intensive. To dedicate greater computer resources to the problem, this thesis proposes a way to distribute the problem over many different computers using the Berkeley Open Infrastructure for Network Computing (BOINC), an open-source platform where people can volunteer their personal computer to work on various problems while their computer is idle. My program runs a genetic version of a gradient descent algorithm, including conjugate gradient methods, that runs the algorithm in parallel on many computers at once to find a solution faster and to avoid some common problems of gradient descent, such as getting trapped in local minima.

DARWINIAN GRADIENT DESCENT
by
Michael Levin

ABSTRACT

Large optimization problems of many variables can be difficult to solve and very computationally intensive. To dedicate greater computer resources to the problem, this thesis proposes a way to distribute the problem over many different computers using the Berkeley Open Infrastructure for Network Computing (BOINC), an open-source platform where people can volunteer their personal computer to work on various problems while their computer is idle. My program runs a genetic version of a gradient descent algorithm, including conjugate gradient methods, that runs the algorithm in parallel on many computers at once to find a solution faster and to avoid some common problems of gradient descent, such as getting trapped in local minima.

ACKNOWLEDGEMENTS

I would like to thank Dr. Gregory Bard, Dr. Michael Black, and Dr. Angela Wu for their assistance with the project. I would also like to thank Genna Duberstein for her support and help with illustrations and Divya Abhat for proofreading the final draft.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

LIST OF TABLES vi

LIST OF FIGURES vii

CHAPTER

1. Introduction 1

2. Gradient Descent Background 5

   Conjugate Gradient Methods 8

   Step Search 10

   Explicit and Induced Derivatives 12

   Complexity 13

3. Volunteer Computing Background 15

   Advantages and Disadvantages 15

   BOINC 17

   Software Engineering Aspects of BOINC 18

4. Speculation 20

5. The Genetic Algorithm 23

   Population 23

   Fitness Function 23

Selection Criteria 24

Crossover 24

Mutation 25

Observations 25

6. Setting up the BOINC Application 27

Work Manager 27

Work Generator 28

Assimilator 29

Entry Point for Gradient Descent 29

7. Problems 31

The GPS Problem 31

The Apollonius Circles Problem 32

The Apollonius Spheres Problem 34

Brent Equations 36

8. Results 37

Speculation 37

Avoiding Local Minima 39

BOINC 41

9. Conclusions 43

APPENDIX

A. Error Bounds On Johnson Derivative Formula vs. Definition of Derivative 45

REFERENCES 49

LIST OF TABLES

Table Page

1. Describes the calculations of the conjugate of β for the gradient methods. 9
2. Percent of success and mean, median, and standard deviation of successes for GPS problem. 38
3. Percent of success and mean, median, and standard deviation of successes for Apollonius circles problem. 38
4. Percent of success and mean, median, and standard deviation of successes for Apollonius spheres problem. 39
5. Timing results for Apollonius spheres problem. 39
6. Attempting to avoid local minima using a constant step search 40
7. Attempting to avoid local minima using Newton's method with Armijo's rule 41

LIST OF FIGURES

Figure	Page
1. A visual representation of gradient descent.	6
2. A graph of the function $f(x) = x^3 - 3x^2$.	8
3. Geometric depiction of the GPS problem.	33
4. Geometric depiction of the Apollonius circles problem.	33
5. Geometric depiction of the Apollonius Spheres problem.	35
6. A graph of the function $f(x) = x^4 - 3x^2 + 2x$.	40
7. Average time of BOINC computations and overhead.	42

CHAPTER 1

Introduction

Gradient descent is an approximation algorithm used in optimization problems to find minima in differentiable functions of several variables. This is done by exploiting a very convenient property of the gradient. Since the gradient vector always points in the direction of steepest increase of a function at any given point, the negative of the gradient always points in the opposite direction, or the direction of steepest decrease. For this reason, Augustin Cauchy originally referred to the method as "The Method of Steepest Descent" (Cauchy, 1847). By continually taking steps in the direction of the negative gradient until the gradient is zero, gradient descent is able to find minima in a function. The components of the gradient vector consist of the partial derivatives of the function, written as

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_n} \right). \tag{1.1}$$

One convenient property of the gradient descent algorithm is that it can be adapted to solve systems of equations. As either the number of equations and unknown variables becomes large in complex problems, gradient descent becomes an interesting tool to consider for finding a solution, especially when all that is really known is the function itself. However, this tool also has a number of problems. For instance, gradient descent depends heavily on the initial conditions provided. The

algorithm must be provided some initial "guess" starting point, which ultimately determines what minimum (if any) the algorithm will find. In addition, there is no straightforward way to verify whether a solution that the algorithm has converged to is a global or local minimum.

In addition to simply stepping in the direction opposite the gradient, the algorithm can be adapted to use a few complementary methods to determine a search direction, referred to as conjugate gradient methods. Instead of simply stepping in the direction of the negative gradient, these methods add another term when calculating the next search direction that is calculated by multiplying a coefficient β by the current search direction. Although all of the different conjugate gradient methods differ in how β is calculated, the idea is the same. The extra term can be thought of as giving the algorithm an extra momentum in the direction of the current search direction. This is very helpful in getting the algorithm to converge to a minimum in fewer steps and possibly even skipping over some small local minima along the way.

One of the experiments performed involved speculating over gradient descent and the different conjugate gradient methods, which involved trying each method at each step and picking the one that resulted in the best (lowest) point for that iteration. Speculation is useful because it ensures that if some method is much better for a given problem, it will be used more often. The results were then compared to all of the other methods used by themselves, in addition to a baseline where a random method was selected for each iteration. Interestingly, it was found that both speculation and choosing a random method gave results comparable with the best methods for that problem used individually. Since selecting a random method is also much less computationally intensive than speculation, which requires each method be tried once each iteration, the random method approach is able to enjoy most of the benefits of speculation at a much lower computational cost.

This thesis then discusses a modification to the gradient descent algorithm by adapting it to work as a genetic algorithm. The algorithm works by generating a large number of points and performing a certain fixed number of iterations of gradient descent with a randomly chosen method. After all of the new points are returned, they are ranked by how "low" they are. In the original version, the bottom 25 percent are tossed out, with new points generated to replace them. This process continues until either a solution is found, some fixed number of different solutions are found, or a fixed number of iterations are reached. In a second version, a mutation element was added for the middle 50 percent of points, where the points were slightly perturbed in different directions by random values within a small range. Although these points are only moved a relatively small distance, this perturbation can sometimes help in getting the point "unstuck" and able to converge to a solution.

Finally, this framework is used to create a program that is distributed over BOINC, a volunteer computing platform, that is able to take advantage of the genetic algorithm aspects while being distributed over many different computers at the same time. Since there are many points being generated per iteration and sent to different computers with instructions to select a randomly chosen method, the final version of the program is able to get around some of the pitfalls of the standard gradient descent algorithm, such as getting stuck in local minima and committing to only one method. This is extremely useful in very large problems where finding an answer is almost impossible with symbolic mathematics or with other numerical approximation methods.

In this thesis, Chapter 2 discusses the mathematical aspects of gradient descent and the associated conjugate gradient methods. It also discusses how different parts of the algorithm work, such as the step search methods, analytic versus induced derivative, and analyzes the complexity of the algorithm. Chapter 3 discusses volun-

teer computing and argues for the use of a volunteer computing platform to address large scientific problems that involve intense computation, such as large optimization problems. It also discusses the BOINC platform specifically and touches upon some of the benefits and problems that one may encounter in setting up a BOINC server. Chapter 4 focuses on the idea of speculation, along with the advantages and disadvantages. It also compares speculation to using only single methods, along with picking a random method at each iteration. Chapter 5 provides a background introduction to genetic algorithms and discusses the role that genetic algorithms play in avoiding some of the more common problems with gradient descent. Chapter 6 touches upon the details and implementation of the distributed algorithm as used over BOINC. Chapter 7 describes the mathematical problems that were used for testing the different approaches used in this thesis. Chapter 8 contains the results found from different aspects of the project, such as speculation, genetic algorithms, and BOINC. Chapter 9 summarizes the project and the conclusions of the thesis.

CHAPTER 2

Gradient Descent Background

This chapter introduces gradient descent and the conjugate gradient methods used in the project. It also describes how gradient descent works in finding a search direction, deciding on a step search to figure out how much to move in that direction, explicit and induced derivatives, and the complexity involved in the problem.

Gradient descent, which is sometimes referred to as the method of steepest descent, is an optimization algorithm that helps you pick a search direction when trying to minimize a function. The equation to find the search direction is

$$\vec{z}_k = -\vec{g} + \beta_k \vec{z}_{k-1}, \tag{2.1}$$

where $\vec{z}$ is the search direction, $-\vec{g}$ is the negative of the gradient, and β is a real valued coefficient representing an added momentum in the old search direction. For gradient descent, the value for β is zero. However, this value is used in the conjugate gradient methods discussed in the following section.

Gradient descent can be used to find minima for everything from multi-dimensional functions to very large systems of equations. It can be thought of as a ball rolling down a hill and finding equilibrium at a minimum. An example is shown in Figure 1. In a system of equations that consists of many equations and unknowns where not much else is known about the problem, it becomes one of the fastest options available for finding a numerical solution. One of the major competitors to gradient

descent is the multi-dimensional Newton's method. However, it becomes impractical for large problems, as it requires inverting large matrices at each iteration. This computation often leads to large rounding errors and can take a very long time to complete. Gradient descent is also very useful in problems where you have a general idea of where the solution should be located. Since gradient descent relies heavily on a starting guess, providing a guess that is close to a solution will usually lead to quick convergence to the solution.

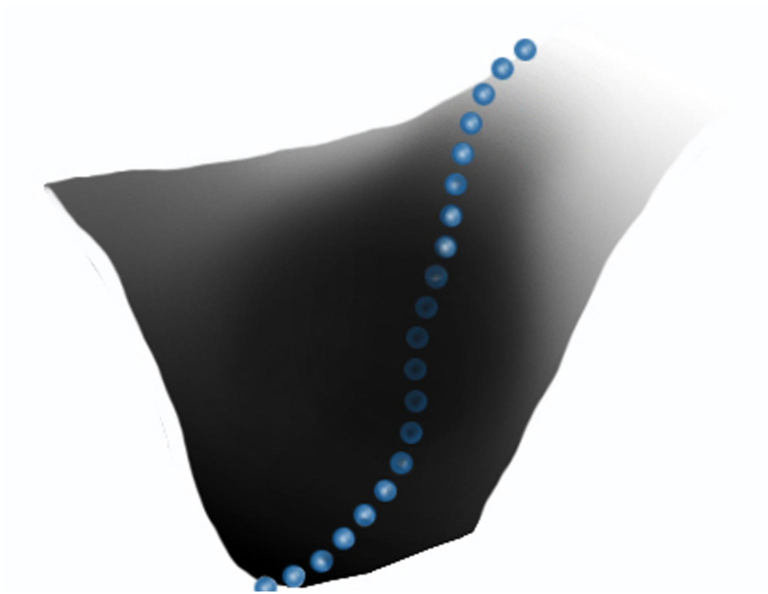

Figure 1. A visual representation of gradient descent.

To minimize a single equation with gradient descent is simple. You take the gradient of that function and take a small step in that direction. If you keep doing this for long enough and you choose a fairly good initial guess, you are able to minimize the function (so long as there is a minimum). However, using gradient descent to find a solution to a system of equations requires a slightly different procedure.

Imagine, you have a system of equations, such that

$$f_1(x_1, x_2, ..., x_n) = 0,$$
$$f_2(x_1, x_2, ..., x_n) = 0,$$
$$\vdots$$
$$f_m(x_1, x_2, ..., x_n) = 0.$$

Therefore, we are looking for a vector $\vec{x} = (x_1, x_2, ..., x_n)$, such that it is a solution to the system of equations above. Imagine a function s, such that

$$s(x_1, x_2, ..., x_n) = \sum_{i=1}^{m} f_i(x_1, x_2, ..., x_n)^2. \tag{2.2}$$

The function clearly must always be non-negative, since it is a sum of squares. Furthermore, we know that the solutions of the system of equations must make $s(\vec{x})$ zero, since all of the individual functions are zero. Furthermore, if any of the equations is non-zero, $s(\vec{x})$ must be positive. Therefore, we know that by minimizing the function $s(\vec{x})$, we have found a solution to the system of equations above. We refer to the solution evaluated at $s(\vec{x})$ as the residual norm. In this example, as well as all of the experiments in this project, the norm used is the L_2 norm.

However, there are a number of problems that come up when using the standard gradient descent algorithm. One of these is that gradient descent is highly dependent on an initial guess. Given a bad guess, the algorithm might send you in a direction for which it will never converge. For instance, observe the function $f(x) = x^3 - 3x^2$ displayed in Figure 2.

If your algorithm decides to select $x = -1$ as a starting point, there will never be convergence. The algorithm will simply continue descending down to $-\infty$, or preferably until a stop condition is reached and the algorithm is aborted. Similarly, the algorithm might converge to a local minimum, when one presumably would be

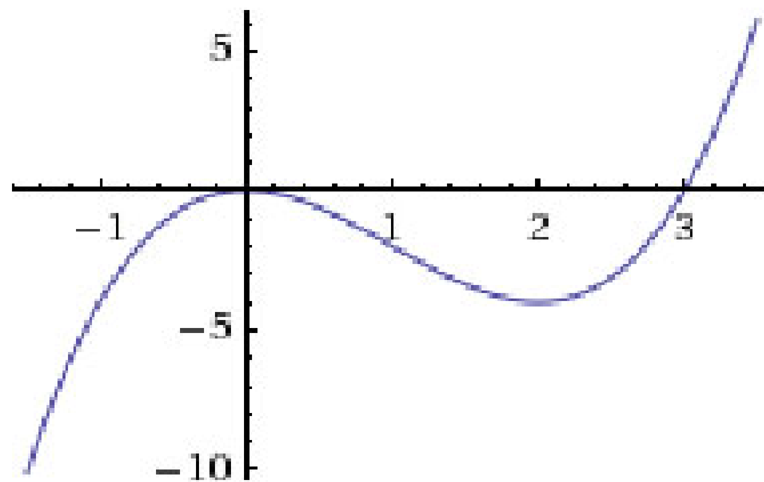

Figure 2. A graph of the function $f(x) = x^3 - 3x^2$.

looking for a global one. There is no natural way to determine whether the solution provided by gradient descent, or any of the conjugate methods described below, has converged to the global minimum or one that is merely local.

Another problem can arise with the propagation of round-off error in the floating point arithmetic. Since the gradient descent algorithm can take thousands or even many billions of iterations to converge to a solution in larger problems, a very small round-off error in each computation might become a very large one as the algorithm completes.

One possible solution to this problem is the use of an arbitrary-precision-arithmetic library, such as the Multiple Precision Floating-Point Reliable library (MPFR). However, the use of such a library, while greatly reducing the problem of round-off errors, greatly increases the total running time of the gradient descent algorithm. An important trade-off must ultimately be made between the accuracy needed in the solution and the amount of time it will take to reach the solution.

Conjugate Gradient Methods

Table 1. Describes the calculations of the conjugate of β for the gradient methods.

Conjugate Gradient Method	Calculation for β
Gradient Descent	$\beta_k = 0$
Fletcher-Reeves	$\beta_k = \frac{<\vec{g}_k, \vec{g}_k>}{<\vec{g}_{k-1}, \vec{g}_{k-1}>}$
Polak-Rebiere	$\beta_k = \frac{<\vec{\gamma}_k, \vec{g}_k>}{<\vec{g}_{k-1}, \vec{g}_{k-1}>}$
Sorenson-Wolfe	$\beta_k = \frac{<\vec{\gamma}_k, \vec{g}_k>}{<\vec{\gamma}_k, \vec{z}_k>}$
Hestenes-Stiefel	$\beta_k = \frac{<\vec{g}_k, \vec{\gamma}_k>}{<\vec{z}_{k-1}, \vec{\gamma}_k>}$

The conjugate gradient methods were first developed in the the second half of the 20th century (Hestenes, 1952 and Fletcher 1963). They build on the idea of gradient descent by adding another term to the equation. This term, $\beta_k \vec{z}_{k-1}$ can be thought of as a momentum in the current search direction. The coefficient that determines this momentum is β and is determined differently for each method. Table 1 shows the different ways that this coefficient is calculated for some of the different methods. In the table, $\vec{z}$ is the search direction, $\vec{g}$ is the gradient, and $\vec{\gamma} = \vec{g}_k - \vec{g}_{k-1}$, or the change in the gradient.

The conjugate gradient methods are most effective and far superior to simple gradient descent when the Hessian matrix is not numerically close to the identity matrix. In fact, the further away from the identity it is, the better the conjugate gradient methods perform. In all of the problems for which experiments were performed, the conjugate gradient methods were vastly superior to gradient descent. Specific results are discussed in Chapter 8.

They are also effective for avoiding local minima and converging to global ones. While this is not always the case, the momentum given to the conjugate gradient methods by the extra β term is sometimes successful in having gradient descent skip over tiny local minima entirely. An analogy suggested by Dr. Gregory Bard is that while gradient descent can be thought of as a tiny pebble rolling down a hill to get

to the bottom, the conjugate gradient methods are more like a boulder that races down the hill fast enough that it can roll through some local minima without getting stuck. Such an example is shown in Chapter 8.

Another interesting case to mention is how gradient descent and the conjugate gradient methods behave around saddle points. While a saddle points is not considered a local extremum, mathematically $\vec{g} = \vec{0}$. This means that there is a chance that the algorithm might terminate under the assumption that it has found a local minimum. However, either the conjugate gradient methods, or a simple mutation in the distributed version can get you out of a saddle point.

Step Search

While gradient descent determines the direction in which we want to move, it is important to also figure out by how much we want to step in that direction. More specifically, we are looking for a λ, such that

$$f(\vec{x_k} + \lambda \vec{z_k}) < f(\vec{x_k}), \tag{2.3}$$

where solving for λ becomes just a univariate optimization problem.

There are many common ways to deal with this problem. One simple way is just to fix a very small constant ϵ, for instance $\epsilon = 0.005$, and to use this step size for each iteration of the problem. Another common method is to use the Newton's method, which is a very efficient way to converge to the roots of functions when selecting a point "sufficiently" close to the root. With an adjustment using Armijo's rule, we can ensure that Newton's method will converge to a root regardless of the initial guess. Both of these methods will be discussed in this section, as well as their advantages and disadvantages.

Using a constant step search has a number of advantages and disadvantages.

The obvious advantage is that it is rather fast, as it basically eliminates the need for almost any significant computation. It is also a very easy method to implement programatically. However, it generally leads to one's programs converging to a solution after many more iterations. A more advanced step search is able to step by larger amounts when there are no minima within a nearby neighborhood. This leads to fewer actual iterations, which in itself limits the amount of computation required since a new search direction does not need to be computed as often. Another potential problem with a constant step search is that the program can keep missing the minimum. Imagine that you are approaching the minimum of the function $f(x) = x^2$ and are at the point $x = 0.05$. Suppose your step search is 0.1. In this case, you will continually jump between the point $x = \pm 0.05$. If your stop condition indicates that you need to be between ± 0.03 of the minimum, the algorithm will simply never converge.

Therefore, it is essential to make a small modification to the constant step search by building in a check to make sure that the actual value of the function decreases for every iteration. For every single time the function doesn't decrease, the constant step search amount is decreased. This is an easy fix to keep the problem from endlessly bouncing around a minimum and to eventually converge.

However, a much more interesting step search is derived from using Newton's method in conjunction with Armijo's rule. Newton's method, as we remember from *Calculus I* is

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)} \tag{2.4}$$

However, the problem with Newton's method is that it only guarantees local, not global convergence. A modification that can be applied to ensure global convergence is called Armijo's rule (Armijo, 1996). Armijo's rule uses the result of the step

search given by Newton's step search and continually averages it with the previous step search amount while the new point is still better than the previous point. For the initial guess, we use -1.

Explicit and Induced Derivatives

In an ideal world, for every function provided, a corresponding derivative would be as well. This would ensure that the derivative is as accurate as possible, as well as simplify the computation needed to be done. However, this implies that the derivatives are found by the programmer's pencil. This is either impossible or impractical in a number of different situations. As already mentioned, we are trying to minimize a function s, where

$$s(\vec{x}) = \sum_{i=1}^{i=m} f_i^2(\vec{x}). \tag{2.5}$$

Therefore, taking a derivative, we get

$$\frac{ds}{dx_j} = \sum_{i=1}^{i=m} 2 f_i(\vec{x}) \frac{df_i}{dx_j}. \tag{2.6}$$

Therefore, there will be mn partial derivatives of the form $\frac{df_i}{dx_j}$. The matrix containing these values is referred to as the Jacobian matrix. Take, for example, the Apollonius Circle problem, which attempts to find a circle that is tangent to three other circles in a plane. The system of equations consists of 9 equations and 9 unknown variables. In order to explicitly write out the complete derivative, there must be 81 different partial derivatives provided.

As the size of the problem grows, even with problems where the derivatives can be easily calculated, explicitly writing out the derivative can become difficult very quickly. Luckily, there is a simple solution. An induced derivative, such as

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h} \tag{2.7}$$

can be used by fixing a very small value for h. A much better variation on this equation,

$$f'(x) \approx \frac{4}{3}\left[\frac{f(x+h) - f(x-h)}{2h}\right] - \frac{1}{3}\left[\frac{f(x+2h) - f(x-2h)}{4h}\right] \tag{2.8}$$

is the one used in this project, and is referred to as either the Johnson Derivative Formula or the five-point derivative approximation (Cheney, 2008). The value $h = .00001$ was used in these experiments. An analysis of the relative errors of the two methods is provided in Appendix A.

Complexity

The complexity of gradient descent from start to finish is difficult to analyze. Since gradient descent, along with the conjugate gradient methods, are very sensitive to both the starting conditions and the function under consideration, this task is basically impossible. However, it is possible to analyze the complexity of a single iteration of the algorithm in both space and time.

We will adopt the convention where m represents the number of equations and n represents the number of unknowns. In terms of space, the complexity is $O(mn)$ if the induced derivative is used and $O(mn)$ otherwise. In the case of the induced derivative, the main vectors kept track of are the current guess, gradient, previous gradient, current search direction, and old search direction. The size of these are determined by the number of variables. However, when an explicit derivative is used, we are now forced to keep track of the Jacobian matrix, which is of size mn. Therefore, the space complexity becomes $O(mn)$.

The time complexity is slightly more complex, since there are a number of different components that need to happen through each iteration. The components include: calculating the gradient, calculating the beta, finding the new search direction, calculating the step amount, and updating the guess. Calculating the gradient is an $O(mn)$ operation for both the explicitly defined derivative and the induced derivative. When the derivative function is provided explicitly, first the Jacobian matrix must be computed, which is an $O(mn)$ calculation. After this is done, it is simply an $O(n)$ operation to fill in the gradient vector. Using the induced derivative is an $O(mn)$ computation as well. Calculating the beta is an $O(1)$ computation as long as we don't need to compute the Hessian. Finding the new search direction is an $O(n)$ operation, since the new direction needs to be applied to all of the unknowns in the problem. The step search varies depending on the method. The time complexity of the fixed step amount is obviously $O(1)$. The Newton step search is $O(m)$, since it needs to calculate the value of a function at a particular point. Changing the guess by moving by the step size in the new search direction is an $O(n)$ operation, since it needs to be applied to all of the unknowns or dimensions of the problem. Therefore, the overall time complexity becomes $O(mn)$.

CHAPTER 3

Volunteer Computing Background

Volunteer computing is a system where people all over the world can support projects by allowing their computers to work on research problems while their computer is on but is not being used. This paradigm leads to many interesting advantages and disadvantages over similar distributed paradigms such as cloud computing or cluster computing.

Advantages and Disadvantages

One of the biggest advantages to volunteer computer is cost. For instance, to set up a BOINC project, the biggest barrier to entry is a powerful server that will run BOINC server and connect to the client computers. Once that is set up and the project is configured properly, most of the large expenses of using a large distributed system solving difficult problems is absorbed by the volunteers, since they are donating both their personal computer power and electricity to help solve your problem. These are two major costs that a project would otherwise need to absorb up front before any of the research could be performed. For instance, assume the computing power is purchased from Amazon's Elastic Cloud (EC2) and it is a fairly big project that requires about 10,000 computers (or 100 TeraFLOPS– a realistic number for a medium-sized BOINC project) running in parallel. The cost of this much computer power through EC2 would amount to about $43.8 million.

The equivalent for a BOINC server would be about $200,000 to buy the server and one or two employees to support it (Anderson, 2010). This is a tremendous savings and worthy of consideration by any new projects in need of large computing power.

However, there are a number of disadvantages to keep in mind as well. Since the computers are only donated in their idle time, they are not nearly as reliable as purchasing the computing time yourself or from a third party such as Amazon. It is very hard to predict when and how many computers will be providing time to work on your problem. Volunteers could potentially decide to switch projects, or simply uninstall BOINC. Furthermore, the process of advertising your project to the general public and convincing them that it is a worthwhile project for them to support requires significant effort and marketing knowledge as well.

This is one of the reasons there are so few volunteer projects available. A few of the more popular ones are Internet Mersenne Prime Search (Mersenne), SETI@HOME ("Seti@home"), and FOLDING@HOME (Pande). The Internet Meresenne Prime Search was one of the first major volunteer computing projects. The project started in 1996 and has found the second largest known prime. SETI@HOME was the first major BOINC project. It is a project run out of Berkeley and is used to try to detect intelligent life outside of Earth by analyzing radio signals. FOLDING@HOME is another big project using BOINC that attempts to simulate the way proteins fold and unfold in order to help understand and fight major diseases. It is estimated that there are only about 60 such projects around today, where surely many times more could benefit from a volunteer computing project (Anderson, 2010).

One of the major reasons that volunteer computing is so under-utilized involves the fact that many of the scientists that would code projects and set up a BOINC server are not the type of people that would be good at advertising their project to the world and convincing people to donate their computer resources to help. Another

reason is that there are still a very limited number of people who have heard of platforms such as BOINC or would even be willing to take the time to donate their computer resources to outside projects. In addition, there are security implications, since you are trusting the project to execute code on your computer that may be malicious. As more people become aware of volunteer computing and middleware becomes integrated with already existing popular projects, the number of scientists utilizing volunteer computing for their research hopefully will increase.

BOINC

BOINC is one of the most popular middleware tools for performing volunteer computing. It is an open-source project and is used for most of the volunteer projects running today. It is also the middleware that was used for this project.

The project started out from a 2002 grant from the National Science Foundation. The first projects launched in 2004; there are approximately 60 projects today.

BOINC is set up in a way such that a BOINC server, which is managed by a project administrator, sends work units to client computers with some computation to perform. When the client computer is idle, it processes this work unit and sends it back to the server. The server is then responsible for reading in the input and doing something meaningful with the result.

There is also a reward system built into BOINC. A user is rewarded for her work by being assigned “credits” by BOINC. The more work the volunteer performs, the more credit she will receive. This serves as an engaging way for individuals, passionate about the work they are helping with, to see the impact they make on the project. However, with BOINC there is also a risk of individuals not running the actual algorithm and sending back junk responses to get actual credit. Although this is fairly uncommon, it is worth noting, since it’s a concern that would simply

not arise if, for instance, computer time was purchased through Amazon's EC2 or if a personal cluster was used.

Advertising one's project with BOINC becomes an even bigger issue, since there are almost 60 different projects to choose from. The chances that one's project will be chosen by someone at random is almost non-existent, since some of the bigger projects already have fairly good name recognition. Therefore, it is useful to create a website for the project to explain the purpose and give potential volunteers a reason to become excited and "spread the word."

One interesting feature of BOINC that helps counteract this problem is the ability for the user to connect to more than one project at the same time. Furthermore, the user can also determine the percentage of resources each project can consume. This is important to allow for many big projects to prosper, but projects must still advertise their merits to users to convince them to join.

Result validation is another major hurdle with BOINC. If a project becomes very big and has many volunteers, the chances of someone malicious appearing and trying to sabotage the results increases. One way to get around this is by sending the same point and method to multiple computers at the same time and making sure the result is similar. However, this slows down the process and different floating point calculations among different architectures could lead to the failure of this method. Another method to subvert a malicious user is by supplying a very large population of work units. As long as there are enough other computers connected to the project, the large number of points almost guarantees that more than one of the points will arrive at a solution.

Software Engineering Aspects of BOINC

One of the major difficulties with BOINC is that many people encounter challenges in setting up a project. There are dozens of different dependencies among

required software packages, with some needing to be configured in a certain way for BOINC to run. For instance, if mySQL (one of the dependencies) is assigned an administrator password, BOINC is unable to run. All of these small details are very poorly documented (if at all) on the BOINC website.

Another interesting point is that the BOINC code needs to be written in a certain way for different computer architectures. Therefore, different sets of code need to be written depending on whether the BOINC client is run from a 32-bit or 64-bit computer. This is an important point to note, since it is important for the volunteer to be able to assist the project no matter what kind of computer they are running. This means that one's project must be able to compile in almost every possible architecture. To BOINC's credit, they do have different versions of the BOINC client for most of today's operating systems.

CHAPTER 4

Speculation

This chapter discusses speculation and how it compares to using the different gradient descent and conjugate gradient methods individually. It also compares these to a baseline, which involves picking a method at random at each iteration of the algorithm.

The idea behind speculation arises out of the fact that there are 5 different methods, including the simple gradient descent method, and very little information is available to know which one is best. Given a random problem from a problem set, we want to get the best average performance while performing the least amount of computation on the problem. We do know that if the Hessian is not like the identity matrix, then the conjugate gradient methods will be better than the simple gradient descent. However, the computation of the Hessian does not tell us which conjugate gradient method will be the best. While the calculation of the Hessian can be beneficial in a number of ways, it is important to keep in mind that it is an expensive computation. In all of the experiments that were run, gradient descent has been consistently worse than the conjugate gradient methods. However, the performance of the conjugate gradient methods differed depending on the problem.

The following is a fundamental question: suppose you pick a problem at random and decide to try out all of the different methods. Every method will result in a

different number of iterations to converge to a solution, given the same initial starting point. However, suppose that before each iteration, you have prior knowledge as to which method will result in the best point after that particular iteration and decide to simply choose that method for that particular iteration. Will the number of iterations be lower than simply sticking to the best method throughout?

It turns out that, at least in the problems that are considered here (results are discussed in Chapter 8), the number of iterations were on average very similar to the best method used individually. This is very surprising, since if you select the best method at every iteration, you should never do worse than the best method used all the time. However, one does see speculation performing slightly worse than the best method in a number of different problems. Perhaps the problem is due to the way β is calculated. Since the calculation relies on the previous iteration, an assumption made in the derivation of these formulae might be broken with the conjugate gradient methods relying on the previous iteration. In other words, they depend upon the previous step being calculated using the same method.

However, prior knowledge is computationally very costly. Every method needs to be performed per iteration, along with the accompanying step search. This makes each iteration a factor of about five times slower, since there are four conjugate gradient methods plus ordinary gradient descent. This would be a major performance hit for large problems involving many iterations.

We then analyzed a baseline situation, where a random method is chosen at each iteration. Interestingly, we found that the results were comparable to that of speculation. This means that we can get most of the benefits of speculation without the computational costs. The best method need not be known to get the same quality of results (in terms of the number of iterations). The results of the all of the speculation experiments are displayed in Chapter 8.

For the distributed algorithm, speculation took the form of simply choosing a randomly assigned method and sending it to each computer along with the initial point. This is ideal since it is not more computationally intensive than needed (only one search direction and step search is calculated), but also performs well in trying to minimize the number of iterations without depending too highly on one particular method, as shown in my tests. Doing this is considerably better than simply selecting a method and hoping that it is the best one for a given problem.

CHAPTER 5

The Genetic Algorithm

This chapter provides a brief background on how genetic algorithms work and how the concept can be applied to gradient descent algorithms. Genetic Algorithms are based on Darwin's ideas of natural selection and are a very useful way to finding solutions to difficult problems.

Population

The first step in a genetic algorithm requires randomly generating some large number of potential solutions, commonly referred to as a population. With gradient descent, every member of the population is an array with randomly generated floating point numbers symbolizing an initial vector. The size of the array is the number of variables in the problem being solved. In this research, the randomly generated floating point numbers are limited within the range $(-10^4, 10^4)$. The initial population was set to $1,000$ vectors. These values should be modified to make sense with the problem under consideration.

Fitness Function

Once the initial population is in place, there must be some way to analyze each individual population member (element) to determine how good each one is in relation to all of the other ones. In other words, we need some function that is able to

sort the entire population by the desirability of the elements. There is a very natural fitness function for gradient descent, which just ranks the solutions by the residual L_2 norm. Since we know that the closer this number is to zero, the closer we are to a potential solution, it becomes a very effective tool for integrating the original gradient descent algorithm into a genetic framework. The biological equivalent of this is selection of the fittest.

Selection Criteria

A lifespan of a genetic algorithm usually spans many different generations. Once the initial population is ranked according to the fitness function, a decision must be made determining which elements to keep and pursue further and which elements should simply die off. In my research, a cut-off value of 25% was used, where the top bottom 25 percent are removed and the top 75 percent are maintained. Since we want to keep populations a certain size to keep getting the benefits of a genetic algorithm, the 25 percent that were removed are replaced with new randomly generated elements.

Crossover

Another strategy that can be used and is often effective is crossover. This means combining two different good solutions together in the hopes that the combination might be better. In genetic descent, this might look like creating a new point from either taking the average of two different points from good potential solution in all dimensions, a geometric midpoint, or randomly picking values for each dimension from already existing good potential solutions. The biological equivalent of this is sexual reproduction between two elements. My research found that this is not the most effective strategy. The results are shown in Chapter 8.

Mutation

The last element is mutation, where each of the elements has a random chance of being changed in some way. The biological equivalent to this idea is small mutations in our DNA, which can be caused by cosmic radiation or other environmental factors. For gradient descent, this consists of applying small perturbations to different points. This is particularly effective to do for solutions that have already been found, but are suspected of being local minima. Once the solution is noted, it is often enough to apply a small perturbation in all of the variables (dimensions) of the problem and allowing the algorithm to run on the next point again. The mutation amount, like most parameters of a genetic algorithm, is generated randomly.

Observations

Once the fitness function is applied and the selection, crossover, and mutation steps are complete, we end up with an altered and hopefully superior population in terms of the fitness function. The process then repeats again for further refinement. For each generation, a certain number of points will return with solutions. The algorithm can either terminate at that point, or keep going to try to pick up more solutions depending on what makes sense for the problem. For each element in the generation, about $10,000$ iterations of gradient descent are performed. This number works well for very large problems, but makes less sense for simpler ones. It should be changed, as most of the parameters in gradient descent, depending on the given problem.

In this research, we found that the gradient descent algorithm is very well-suited to be adapted to a genetic algorithm. The genetic components help to overcome some of the serious limitations of gradient descent. For instance, genetic algorithms are great for finding multiple solutions. In the program, the algorithm could be run until one solution is found, a fixed number of solutions are found, or for a fixed

number of generations. Although this cannot determine for certain which solution (if any of them) is the global minimum, it can tell you which ones are not. While not ideal, this is definitely useful information. Similarly, because gradient descent is very reliant on initial conditions, starting with a large number of points distributed over the problem space is effective in raising the probability that a solution (if one exists) will be found.

However, one of the greatest advantages is being able to move the genetic algorithm over to a distributed environment, such as a volunteer computing platform discussed in the next chapter. In such a scenario, every member of the population can be sent off to a separate computer for processing. Therefore, every generation takes about the time of the processing of one element, plus extra overhead. This leads to a significant speed improvement, while maintaining all of the advantages of using the genetic algorithm adaptation described above.

CHAPTER 6

Setting up the BOINC Application

This chapter focuses on the different components that allow for BOINC to work with the gradient descent program. This chapter also describes how these components are implemented to allow for the different genetic aspects to function as we discussed earlier in Chapter 5. These components include the work manager, work generator, assimilator, the client entry point to the gradient descent program and, of course, the actual gradient descent program. These components collectively handle the two main parts of the BOINC application: creating and managing work units on the server and running the gradient descent algorithm on the client computer.

Work Manager

The work manager is responsible for managing the individual work units for the project. The first thing it does is initialize a random initial population of work units. A work unit is simply a file that contains numbers representing some information about the point under consideration for which the gradient descent algorithm will be run. These files are contained within the population directory.

The first three numbers represent the "quality," "generation," and the number of variables in the problem. As the algorithm begins and work units are returned, the quality is determined by the residual L_2 norm of the particular point. The closer this quality number is to zero, the more "fit" the point is considered. This is the value

used for sorting all of the points after each generation and determining and keeping the best ones, removing the worst ones, and applying random mutations to the middle 50 percent of points. The generation value keeps track of how many iterations of the genetic algorithm the point has undergone, and is simply incremented in each iteration. The number of variables is important to keep track of because it indicates the number of floating point numbers that follow. These floating point numbers represent values for the unknowns in the problem for which the algorithm will be performed.

Once the initial population is created, the work manager enters an infinite loop where it is responsible for copying the work units from the population directory into the feed directory. It then waits for the work generator and assimilator, which will be described in the following two sections, to take the population from feed, submit it to the client for processing, and finally take the results and store them in the results directory. Once the results are finally received, they are copied to the processing directory, where each work unit is then inspected, and depending on its quality, is either copied to the junk directory and discarded (bottom quarter of the population in quality ranking), copied back into the population directory as is (top quarter of the population in quality ranking), or randomly mutated through a slight perturbation or having a random value being assigned to one of the variables (middle half of the population in quality ranking). Of the middle half of points that are candidates for mutation, one third are perturbed, another third have a random value assigned to one of the variables, and the final third are left unchanged.

Work Generator

The work generator is responsible for interfacing with the BOINC server and submitting the work units that are placed in the feed directory by the work manager. It goes through each individual work unit, creates a job for it, and registers that job

with the BOINC server so that it is ready to be sent out to the client for processing. This file was based entirely off of the samples that BOINC provides, with no real modifications being necessary.

Assimilator

The assimilator is responsible for getting the results from BOINC after the actual gradient descent work is complete and placing them in the results directory. If there is an error and they can't be successfully copied, a log entry is created in the errors file in the results directory. Once the results are obtained and placed in the results folder, the work manager can take over once again. This assimilator file, much like the work generator, was based on the samples provided by BOINC with no modifications necessary.

Entry Point for Gradient Descent

This file is the primary interface between BOINC and the gradient descent code that was written for this project. It is the code that is run on the client computers that are connected to the project. The file takes a work unit as an input and returns an updated work unit, which is then assimilated into the results directory.

The first thing this program does is read in the quality, current generation, and number of variables. It then loops through and stores the values of each of the variables. Once all of this information has been processed, a method must be chosen. The program picks a random number between 0 and the number of conjugate gradient methods and uses that number to determine which of the conjugate gradient methods will be performed. Regular gradient descent was left off, as it performed significantly worse in all of the experiments, as shown in Chapter 8. Another choice that must be made is the norm that will be used for the problem. For this project, the L_2 norm was used throughout all of the experiments. Depending on the problem

being solved, additional parameters, like known values might need to be specified. Once all of this information is provided, it is passed to the gradient descent solving program, which performs $10,000$ iterations of the randomly chosen method. The gradient descent program returns a new quality value, as well as the new point as its output.

Once the new point and quality value is returned, it needs to be stored in a new work unit. Once this new file is created with the new quality, updated generation value, and new values for each of the unknown variables, this program terminates, and the assimilator takes over.

CHAPTER 7

Problems

This chapter presents the four different problems that the thesis algorithms were tested on: the GPS problem, the Apollonius Circles Problem, the Apollonius Spheres Problem, and the Brent Equation. Each of these involves solving a system of equations of increasing complexity, in terms of the number of variables and number of equations.

The GPS Problem

The GPS problem is one that is both simple and practical in the real world. Imagine you are somewhere on Earth, but you are not sure where. However, you have a GPS that knows the position of three satellites and how far it is located from each one. Finding your position becomes a simple problem that can be derived using the distance formula in 3 dimensions and results in the following system of equations:

$$f_1(x_0, y_0, z_0) = (x_i - x_0)^2 + (y_i - y_0)^2 + (z_i - z_0)^2 - d_i^2 \tag{7.1}$$

$$\vdots \tag{7.2}$$

where your position is represented by the point (x_0, y_0, z_0), the position of the i_{th} satellite is represented by the points (x_i, y_i, z_i), and the distances from you to the i_{th} satellite are d_i.

This is a fairly simple problem computationally, involving just 3 equations and 3 unknowns. The unknown variables we are solving for are x_0, y_0, and z_0, or our location. The known variables are the locations (x_i, y_i, z_i) for $i \in (1, 2, 3)$ and distances d_i for $i \in (1, 2, 3)$ of the three satellites, However, although this is a fairly trivial problem for all of the different methods to solve, it gives us a good baseline for tracking performance of what happens as the complexity of the problems increase. Also, since this problem is fairly small, we can explicitly specify the partial derivatives that are needed to compute the gradient, rather than using the numerical or "induced" derivative. In other problems, the number of derivatives to explicitly specify becomes so large that an induced derivative is used.

In a real GPS system, the locations of the satellites are programmed into the unit. These satellites transmit clock signals to the GPS. The GPS itself knows the actual time and is able to subtract the actual time from the time received from the satellites. Using the time elapsed for the signal to reach the GPS and the speed of light, the GPS is able to calculate the distance to each satellite using the simple formula $d = rt$ (Kalman, 2002).

A geometric depiction of the GPS problem is shown below.

While this is a good starting to estimate how this type of problem is solved in reality, there are many real-world details that have been left out. For instance, four or more satellites are usually used instead of three for greater accuracy. Also, the computations need to be adjusted for errors resulting from problems such as impedance of the radio wave while traveling through the atmosphere.

The Apollonius Circles Problem

In the Apollonius circles problem, you are given the location of the center of three circles and their radii. The goal is to find a single circle that is tangent to all three. One interesting part of this problem is that if the circles do not overlap,

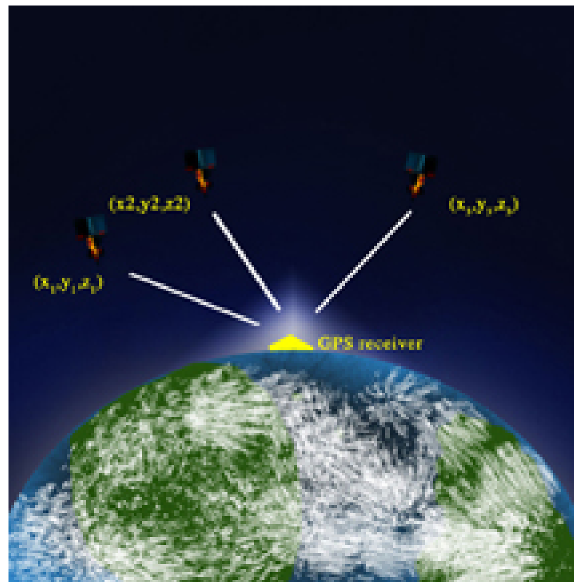

Figure 3. Geometric depiction of the GPS problem.

you end up with eight different solutions that the gradient descent algorithm can converge upon.

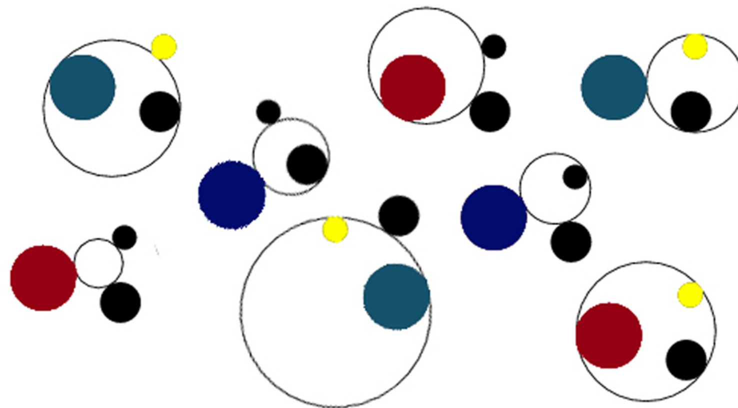

Figure 4. Geometric depiction of the Apollonius circles problem.

The derivation of this problem leads to a system of equations as well, but this time involving 9 equations and 9 unknowns. In this case, we no longer explicitly specify the derivatives, since it would require 81 terms (9 functions must be derived

for each of the 9 variables). Therefore, an induced derivative is used based on the definition of a derivative. The equations for the function f are written below:

$$
\begin{aligned}
f_1(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_A - h_N)(y_A - k_A) - (x_A - h_A)(y_A - k_N) \\
f_2(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_B - h_N)(y_B - k_B) - (x_B - h_B)(y_B - k_N) \\
f_3(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_C - h_N)(y_C - k_C) - (x_C - h_C)(y_C - k_N) \\
f_4(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_A - h_A)^2 + (y_A - k_A)^2 - r_A^2 \\
f_5(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_B - h_B)^2 + (y_B - k_B)^2 - r_B^2 \\
f_6(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_C - h_C)^2 + (y_C - k_C)^2 - r_C^2 \\
f_7(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_A - h_N)^2 + (y_A - k_N)^2 - r_N^2 \\
f_8(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_B - h_N)^2 + (y_B - k_N)^2 - r_N^2 \\
f_9(x_A, y_A, x_B, y_B, x_C, y_C, h_N, k_N, r_N) =& (x_C - h_N)^2 + (y_C - k_N)^2 - r_N^2,
\end{aligned}
$$

where (h_N, k_N) represents the center of the tangent circle, r_N represents the radius of the tangent circle, and (x_A, y_A), (x_B, y_B), and (x_C, y_C) represent the point of tangency between the solution circle and the first, second, and third circle. Those are the unknown variables we are trying to solve for. The known variables are (h_i, k_i) and r_i for $i \in (A, B, C)$.

The Apollonius Spheres Problem

The Apollonius spheres problem is very similar to the Apollonius circles problem, but instead of finding a circle that is tangent to three other circles, you must find a sphere that is tangent to four other spheres. Now, instead of 9 equations and 9 unknowns, the problem becomes computationally much bigger, involving 16 equations and 16 unknowns.

Figure 5. Geometric depiction of the Apollonius Spheres problem.

The system of equations is below:

$$\begin{aligned}
f_1(r_0, x_{c0}, y_{c0}, z_{c0}, x_{1..4}, y_{1..4}, z_{1..4}) &= (x_i - x_{ci})^2 + (y_i - y_{ci})^2 + (z_i - z_{ci})^2 - r_i^2 \\
&\vdots, \\
f_5(r_0, x_{c0}, y_{c0}, z_{c0}, x_{1..4}, y_{1..4}, z_{1..4}) &= (x_i - x_{c0})^2 + (y_i - y_{c0})^2 + (z_i - z_{c0})^2 - r_0^2 \\
&\vdots, \\
f_9(r_0, x_{c0}, y_{c0}, z_{c0}, x_{1..4}, y_{1..4}, z_{1..4}) &= (x_i - x_{ci})(z_i - z_{c0}) - (x_i - x_{c0})(z_i - z_{ci}) \\
&\vdots, \\
f_{13}(r_0, x_{c0}, y_{c0}, z_{c0}, x_{1..4}, y_{1..4}, z_{1..4}) &= (y_i - y_{ci})(z_i - z_{c0}) - (y_i - y_{c0})(z_i - z_{ci}) \\
&\vdots,
\end{aligned}$$

The Apollonius spheres problem is known to have interesting applications in biochemistry, by helping to model the situation where a spherically shaped hormone or drug needs to be fitted within a protein (Lewis, 2003).

where (x_{ci}, y_{ci}, z_{ci}) for $i \in 1, 2, 3, 4$ represents the center of the four spheres and r_i for $i \in 1, 2, 3, 4$ are the radii. This information is known. The unknowns are the center of the tangent sphere (x_{c0}, y_{c0}, z_{c0}), its radius r_0, and the four points (x_i, y_i, z_i) for

$i \in 1, 2, 3, 4$, which represents the points of tangency between the solution sphere and the first, second, third, and fourth sphere.

Brent Equations

The Brent Equations are a system of cubic equations, whose solutions can be exploited in order to create a very fast matrix multiplication algorithm. By solving an equation of 594 unknowns and 729 equations exactly, you can solve a 3x3 matrix multiplication in 22 steps, as opposed to 27 steps in the naive algorithm. A solution to a system of equation with 4096 equations and 2304 or 2256 unknowns would lead to the fastest algorithm for the multiplication of a 4x4 matrix. In general, there are n^6 equations and $3n^2$ variables to compute an n x n matrix product in s-steps. This yields a $\theta(N^{\frac{log(s)}{log(n)}})$ algorithm for n x n matrices as the $\lim_{n\to\infty}$ (Bard and Brent, 1970). Since gradient descent is an approximation algorithm, although a solution to the smaller cases can be found using the algorithm, the solutions found will only lead to an approximate solution.

The Brent Equations can be represented simply as a summation over 6 different indices:

$$\sum_{k=1}^{k=s} \alpha_{wxk}\beta_{yzk}\gamma_{ijk} = \delta_{iw}\delta_{xy}\delta_{jz} \forall \begin{cases} 1 \leq u \leq c & 1 \leq w \leq a \\ 1 \leq x \leq b & 1 \leq y \leq b \\ 1 \leq z \leq c & 1 \leq j \leq c \end{cases}$$

CHAPTER 8

Results

This chapter presents the results of experiments performed in a number of different areas in the project. It compares and contrasts speculation and the random-selection method option to the gradient descent method and conjugate gradient methods. It then talks about the use of the conjugate gradient methods in avoiding some local minima entirely. It then contrasts the results of the genetic algorithm version of the program to the regular version, as well as presents the results obtained with BOINC.

Speculation

The speculation results were the most surprising of the entire project. While it seemed that picking the best result for each method at each iteration (the greedy meta-algorithm) would result in a solution at least as good as the best method on its own, this turned out to not be the case. This might be due to hidden assumptions in the derivations of the conjugate gradient methods that rely on β being calculated using the same method for each iteration. For the results, 100 sample points were randomly picked, scattered around the interval $(-10^4, 10^4)$ in each dimension of the problem. Five random problem initial condition for the problem were generated around the interval $(-10^3, 10^3)$. The different algorithms ran until a solution was found using the GPS, Apollonius circles, and Apollonius spheres problem. The

Table 2. Percent of success and mean, median, and standard deviation of successes for GPS problem.

Algorithm	% Terminate Success	Mean	Median	Standard Deviation
Speculation	100%	58.71	42.5	48.7
Random	100%	81.28	54.5	91.9
Gradient Descent	100%	2982.49	268.5	11690.02
Fletcher-Reeves	99%	2433.82	89	19962.28
Polak-Rebiere	100%	60.41	44	53.07
Sorenson-Wolfe	100%	59.97	44.5	60.4
Hestenes-Stiefel	100%	1361.85	627	1484.93

Table 3. Percent of success and mean, median, and standard deviation of successes for Apollonius circles problem.

Algorithm	% Terminate Success	Mean	Median	Standard Deviation
Speculation	100%	52.61	40.5	52.29
Random	100%	68.88	52	55.1
Gradient Descent	98%	673.29	256	1017.67
Fletcher-Reeves	99%	657.62	80	2250.06
Polak-Rebiere	100%	55.13	42	53.83
Sorenson-Wolfe	100%	52.49	43	32.57
Hestenes-Stiefel	100%	1213.57	574	1352.58

randomly generated points used were the same for each method. The maximum number of steps was set to 3,000 before the stop condition was reached for the GPS Problem; for the Apollonius circles and spheres problem, the stop condition was set at a maximum of 30,000 steps. Table 2, 3, and 4 show the percent of solutions that converged within the designated number of steps, as well as the mean, median, and standard deviation of the results that did manage to converge to a solution.

It is also helpful to compare the time required by each of the methods to complete the 100 random points. Table 5 provides the times it took for the different methods to converge to a solution for the Apollonius spheres problem.

It is interesting to note that speculation takes much less time than gradient descent and Fletcher-Reeves, even though speculation does each of those methods in

Table 4. Percent of success and mean, median, and standard deviation of successes for Apollonius spheres problem.

Algorithm	% Terminate Success	Mean	Median	Standard Deviation
Speculation	100%	138.58	76.5	188.08
Random	100%	250.27	107	392.6
Gradient Descent	89%	2011.52	1058	2816.13
Fletcher-Reeves	100%	818.89	133.5	2866.06
Polak-Rebiere	100%	134.31	78	165.64
Sorenson-Wolfe	100%	144.69	80.5	181.99
Hestenes-Stiefel	100%	3048.47	2719	1798.08

Table 5. Timing results for Apollonius spheres problem.

	Seconds
Speculation	13.442788
Random	5.187284
Gradient Descent	134.463333
Fletcher-Reeves	11.698850
Polak-Rebiere	3.586853
Sorenson-Wolfe	3.527979
Hestenes-Stiefel	3.500077

every iteration. The reason the time is much lower is that speculation converges in fewer iterations than either gradient descent and Fletcher-Reeves.

Avoiding Local Minima

One of the areas of research was to try to examine how conjugate gradient methods can help avoid local minima. For instance, observe the graph in Figure 10 of the equation $f(x) = x^4 - 3x^2 + 2x$.

The graph has two minima, a local one at $x = 1$ and a global one at $x \approx -1.366$. The goal is to start somewhere to the right of $x = 1$ and still get to the minimum located at -1.366. The first test that was run used an initial guess of $x = 30$ using a constant step search of $\epsilon = 0.0005$. The results are displayed in Table 6.

As you can see, Gradient Descent takes you right to the local minimum,

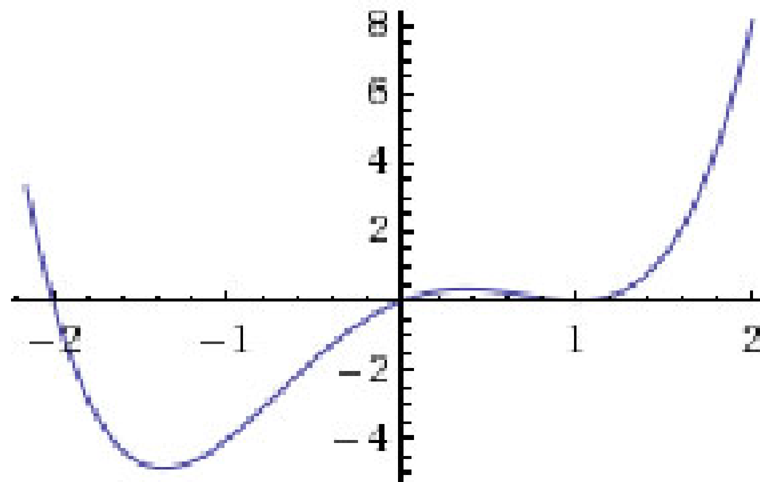

Figure 6. A graph of the function $f(x) = x^4 - 3x^2 + 2x$.

Table 6. Attempting to avoid local minima using a constant step search

Conjugate Gradient Method	Number Of Iterations	Point Converged
Gradient Descent	839	1.03115
Fletcher-Reeves	91	-1.36701
Polak-Rebiere	N/A	Diverges
Sorenson-Wolfe	N/A	Diverges
Hestenes-Stiefel	168	-1.30692

while Fletcher-Reeves and Hestenes-Stiefel manage to avoid the local minimum and converge to the global minimum. In this particular problem, Polak-Rebiere and Sorenson-Wolfe diverged, which might have to do with using the constant step search and starting further away. When a point closer to the local minimum is taken, the two methods converge without any problem.

In Table 7, we do the same experiment, but this time using Newton's method with Armijo's rule.

It seems that using the Newton's Method step search gets us to a solution in much fewer iterations, but it also seems to always converge only to the nearest local minimum. This was true of all of the experiments that were run. Part of this has to do with far fewer iterations being run, which means the conjugate gradient methods

Table 7. Attempting to avoid local minima using Newton's method with Armijo's rule

Conjugate Gradient Method	Number Of Iterations	Point Converged
Gradient Descent	2	1.00000
Fletcher-Reeves	2	1.00000
Polak-Rebiere	2	1.00000
Sorenson-Wolfe	2	1.00000
Hestenes-Stiefel	2	1.00000

have less time to work.

BOINC

We ran the BOINC test with the 2x2 case of the Brent equations, where $s = 8$ and $a = b = c = 2$. This results in a problem with 64 equations and 96 unknowns. When this problem is run under BOINC, most of the computational time is devoted to the problem, rather than to overhead. This is shown in Figure 11, where each work unit took an average of 108.14 seconds with a combined overhead of only about 8.63 seconds. This results in a significant time savings when this problem is run in parallel over many different computers, with each generation taking on average about 116.77 seconds.

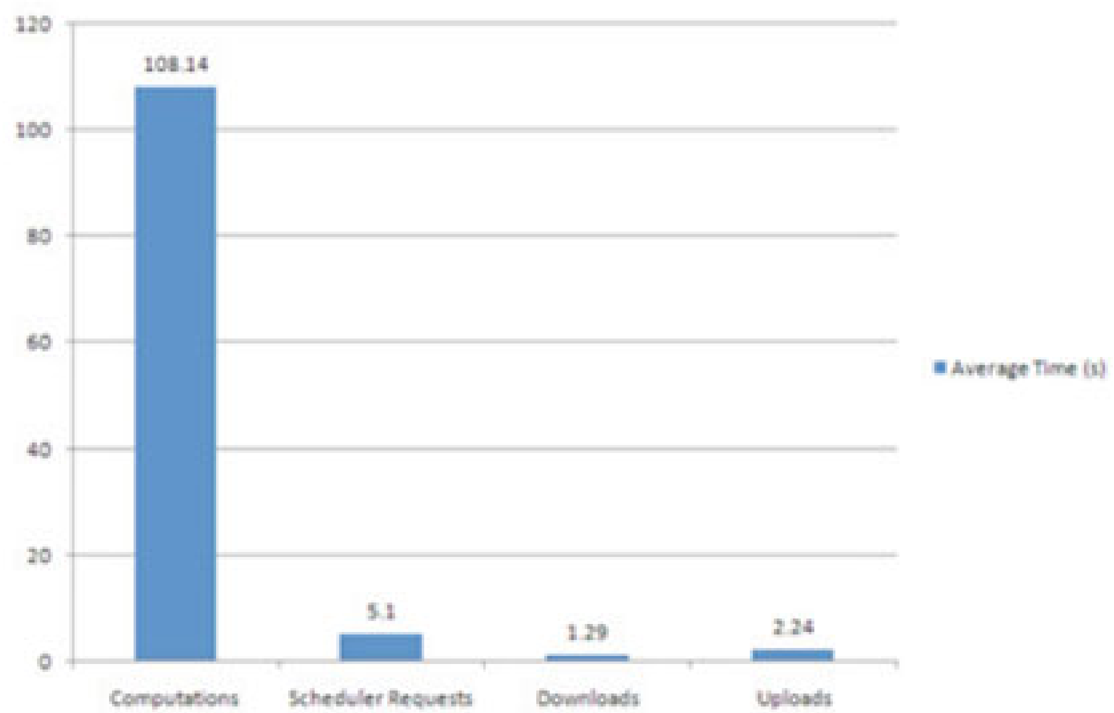

Figure 7. Average time of BOINC computations and overhead.

CHAPTER 9

Conclusions

This thesis presented a method of trying to improve the number of iterations it takes to converge to a solution by speculating over gradient descent and the conjugate gradient methods. Although speculation doesn't always perform better than the best method used alone, it was interesting to discover that picking a method at random at each iteration performs just as well as either speculation or selecting the best method. This is important, since picking a random method for each iteration is not dependent on any one method. If a problem is selected that a particular method is bad for, that unfortunate method will not be the only method being used.

Furthermore, this thesis explored adjusting the algorithm using a genetic approach, which offers the following major improvements over traditional gradient descent:

- The conjugate gradient methods are able to pass over some local minima.

- Having a large starting population of points mitigates gradient descent's dependence on an initial guess.

- The algorithm is able to find multiple solutions and clearly distinguish the solutions which cannot be the global minimum.

This approach was then made to run over BOINC, a volunteer-computing platform, to speed up the genetic algorithm since each generation can be run in parallel.

The BOINC project was coded in an object-oriented, modular way, so that problems can be easily written and plugged into the project. With a little bit more work, this can be adjusted so that researchers from across the country could go to a website and submit a problem. This problem could automatically be injected into the program and sent over the grid to volunteer computers to be solved. Once it is solved, a solution could be sent to the researcher. This eliminates the problem of researchers writing up their own BOINC tools and advertising the project to take advantage of the grid in solving their research problems.

APPENDIX A

Error Bounds On Johnson Derivative Formula vs. Definition of Derivative

The error bounds is very significant in an approximation algorithm such as gradient descent, since a small error in each iteration will propagate and lead to a very big error. Finding the most accurate approximation of the derivative is crucial in keeping the error as small as possible. The analysis in this appendix will show the error of the Johnson derivative formula as being proportional to $h^4 f^{(5)}(x)$, whereas the midpoint derivative formula has an error of $h^2 f'''(x)$.

First, we will examine the error bounds of the formula

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h}. \tag{A.1}$$

To do this, we first work out the following Taylor expansions:

$$\begin{aligned} f(x+h) =& f(x) + f'(x)h + \frac{f''(x)}{2}h^2 + \frac{f'''(x)}{6}h^3 + \frac{f^{(4)}(x)}{24}h^4 + \frac{f^{(5)}(x)}{120}h^5 \\ &+ \frac{f^{(6)}(x)}{720}h^6 + O(f^{(7)}(x)h^7) \end{aligned}$$

and.

$$f(x-h) = f(x) - f'(x)h + \frac{f''(x)}{2}h^2 - \frac{f'''(x)}{6}h^3 + \frac{f^{(4)}(x)}{24}h^4 - \frac{f^{(5)}(x)}{120}h^5 + \frac{f^{(6)}(x)}{720}h^6 + O(f^{(7)}(x)h^7)$$

By evaluating $f(x+h) - f(x-h)$, we get

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{f'''(x)}{3}h^3 + \frac{f^{(5)}(x)}{60}h^5 + O(f^{(7)}(x)h^7) \quad \text{(A.2)}$$

as the rest of the terms vanish. Solving for $f'(x)$, we get

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 - \frac{f^{(5)}(x)}{120}h^4 - O(f^{(7)}(x)h^7). \quad \text{(A.3)}$$

Therefore, we see that the error is proportional to $h^2 f'''(x)$, since that term will dominate.

We will now perform the same analysis for the Johnson derivative formula,

$$f'(x) = \frac{4}{3}\left[\frac{f(x+h) - f(x-h)}{2h}\right] - \frac{1}{3}\left[\frac{f(x+2h) - f(x-2h)}{4h}\right]. \quad \text{(A.4)}$$

We already know from a prior calculation that

$$f(x+h) - f(x-h) = 2f'(x)h + \frac{f'''(x)}{3}h^3 + \frac{f^{(5)}(x)}{60}h^5 + O(f^{(7)}(x)h^7) \quad \text{(A.5)}$$

and we solved that for $f'(x)$ to get

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} - \frac{f'''(x)}{6}h^2 - \frac{f^{(5)}(x)}{120}h^4 - O(f^{(7)}(x)h^7) \quad \text{(A.6)}$$

Solving further, we see that , $\frac{4}{3}f'(x)$ is

$$\frac{4}{3}f'(x) = \frac{4}{3}\frac{f(x+h) - f(x-h)}{2h} - \frac{2}{9}f''(x)h^2 - \frac{f^{(5)}(x)}{90}h^4 - O(f^{(7)}(x)h^7). \quad \text{(A.7)}$$

We now need to perform a similar procedure to determine $f(x+2h)-f(x-2h)$. By the Taylor expansions, we know that

$$\begin{aligned} f(x+2h) =& f(x) + 2f'(x)h + 2f''(x)h^2 + \frac{4}{3}f'''(x)h^3 + \frac{2}{3}f^{(4)}(x)h^4 + \frac{4}{15}f^{(5)}(x)h^5 \\ &+ \frac{4}{45}f^{(6)}(x)h^6 + O(f^{(7)}(x)h^7) \end{aligned}$$

and

$$\begin{aligned} f(x-2h) =& f(x) - 2f'(x)h + 2f''(x)h^2 - \frac{4}{3}f'''(x)h^3 + \frac{2}{3}f^{(4)}(x)h^4 - \frac{4}{15}f^{(5)}(x)h^5 \\ &+ \frac{4}{45}f^{(6)}(x)h^6 + O(f^{(7)}(x)h^7). \end{aligned}$$

Therefore, when solving for $f(x+2h) - f(x-2h)$, we get

$$f(x+2h)-f(x-2h) = 4f'(x)h+\frac{8}{3}f'''(x)h^3+\frac{8}{15}f^{(5)}(x)h^5+O(f^{(7)}(x)h^7)-O(f^{(7)}(x)h^7). \quad \text{(A.8)}$$

Solving for $\frac{1}{3}f'(x)$, we get

$$\begin{aligned} \frac{1}{3}f'(x) = \frac{1}{3}\left[\frac{f(x+2h) - f(x-2h)}{4h}\right] - \frac{2}{9}f'''(x)h^2 - \frac{8}{45}f^{(5)}(x)h^4 \\ - O(f^{(7)}(x)h^7). \end{aligned}$$

Therefore, we now see that,

$$
\begin{aligned}
f'(x) = \frac{4}{3}f'(x) - \frac{1}{3}f'(x) = & \frac{4}{3}\left[\frac{f(x+h)-f(x-h)}{2h}\right] - \frac{2}{9}f''(x)h^2 - \frac{f^{(5)}}{90}h^4 - \\
& \frac{1}{3}\left[\frac{f(x+2h)-f(x-2h)}{4h}\right] + \frac{2}{9}f'''(x)h^2 + \frac{8}{45}f^{(5)}(x)h^4 \\
& - O(f^{(7)}(x)h^7).
\end{aligned}
$$

Therefore, after reducing, we have

$$
\begin{aligned}
f'(x) = & \frac{4}{3}\left[\frac{f(x+h)-f(x-h)}{2h}\right] - \frac{1}{3}\left[\frac{f(x+2h)-f(x-2h)}{4h}\right] + \frac{1}{6}f^{(5)}(x)h^4 \\
& - O(f^{(7)}(x)h^7).
\end{aligned}
$$

We see that the h^2 term neatly drops out and we are left with an error proportional to $h^4 f^{(5)}(x)$ for the Johnson Derivative Formula.

REFERENCES

"Seti@home." http://setiathome.berkeley.edu.

Anderson, David P. (2010, March). "Volunteer computing: The ultimate cloud." *Crossroads* 16(3), 7–10.

Armijo, Larry (1996). "Minimization of functions having lipschitz continuous first partial derivatives." *Pacific Journal of Mathematics* 16(1), 1–3.

Avriel, Mordecai (2003). *Nonlinear Programming: Analysis and Methods.* Dover Publications.

Bard, Gregory. "New practical strassen-like approximate matrix-multiplication algorithms found via solving a system of cubic equations." *In Preparation, Draft Available on Authors Web Page, 2010, http://www.math.umd.edu/bardg*.

Brent, Richard (1970, March). "Algorithms for matrix multiplication." *Tech. Report Report TR-CS-70- 157, Department of Computer Science, Stanford*.

Cauchy, Augustin (1847). "Méthode générale pour la résolution des systéms d'Équations simultanées." *Comp. Rend. Acad. Sci. Paris*, 536–538.

Cheney, Ward and David Kincaid (2008). *Numerical mathematics and computing.* Thompson Learning, Inc.

Fletcher, R. and M. H. D. Powell (1963). "A rapidly convergent descent method for minimization." *The Computer Journal* 6(2), 163–168.

Fletcher, R. and C. M. Reeves (1964). "Function minimization by conjugate gradients." *The Computer Journal* 7(2), 149–154.

Hestenes, Magnus R. and Eduard Stiefel (1952, December). "Methods of conjugate gradients for solving linear systems." *Journal of Research of the National Bureau of Standards* 49(6), 409–436.

Kalman, Dan (2002, November). "An undetermined linear system for gps." *The Mathematical Association of America* 33(5), 384–390.

Lewis, Robert H. and Stephen Bridgett (2003). “Conic tangency and apollonius problems in biochemistry and pharmacology.” *Mathematics and Computers in Simulation* 61(1), 101–114.

Mersenne Research, Inc. “Gimps: Great internet mersenne prime search.” http://www.mersenne.org.

Pande, Vijay and Stanford University. “Folding@home.” http://folding.stanford.edu.