ASYNCHRONOUS GLOBAL OPTIMIZATION FOR MASSIVE-SCALE COMPUTING

By

Travis Desell

Major Subject: Computer Science

Approved by the
Examining Committee:

Carlos A. Varela, Thesis Adviser

Boleslaw Szymanski, Thesis Adviser

Malik Magdon-Ismail, Member

Heidi Newberg, Member

Dave Anderson, Member

Rensselaer Polytechnic Institute
Troy, New York

December 2009
(For Graduation December 2009)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

I would like to thank my family, committee and friends for their kindness, support and understanding. Without them I would never have completed this thesis. I would especially like to thank Carlos Varela for his many years of tutelage, support and advice.

I would also like to thank the Marvin Clan, David Glogau, and the Dudley Observatory for their generous donations to the MilkyWay@Home project, as well as the thousands of volunteers that made this work possible.

This work has also partially been supported by the National Science Foundation under Grant Numbers 0612213, 0607618 and 0448407. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

ABSTRACT

As the rates of data acquisition and cost of model evaluation in scientific computing are far surpassing improvements in processor speed, the size of the computing environments required to effectively perform scientific research is increasing dramatically. As these computing environments increase in size, traditional global optimization methods, which are sequential in nature, fail to adequately address the challenges of scalability, fault tolerance and heterogeneity that using these computing systems entails. This thesis introduces asynchronous optimization strategies which while similar to their traditional synchronous counterparts, do not have explicit iterations or dependencies. This allows them to scale to hundreds of thousands of hosts while not being degraded by faults or heterogeneity. A framework for generic distributed optimization (FGDO) is presented, which separates the concerns of scientific model development, distributed computing and developing efficient optimization strategies; allowing researchers to develop these independently and utilize them interoperably through simple interfaces. FGDO has been used to run these asynchronous optimization methods using an astroinformatics problem which calculates models of the Milky Way galaxy on thousands of processors in RPI's BlueGene/L supercomputer and to run the MilkyWay@Home volunteer computing project, which currently consists of over 25,000 active computing hosts. A simulation environment was also implemented in FGDO, which allowed asynchronous optimization to be examined in a controlled setting with benchmark optimization problems. Results using the simulated environment show that the asynchronous optimization methods used scale to hundreds of thousands of computing hosts, while the traditional methods do not improve or even degrade as more computing hosts are added. Additionally, the asynchronous optimization methods are shown to be largely unaffected by increasing heterogeneity in the computing environment and also scale similarly in a computing environment modeled after MilkyWay@Home. This thesis presents strong evidence of the need for novel optimization methods for massive scale computing systems and provides effective initial work towards this goal.

CHAPTER 1
Introduction

The use of massive-scale heterogeneous computing environments is becoming more widespread through the use of different Internet computing technologies such as the Berkeley Open Infrastructure for Network Computing (BOINC). Volunteer computing infrastructures can offer millions of computing hosts, approaching or surpassing petascale computing power. The low cost and potential computing power of these environments makes them highly desirable to researchers in a broad range of scientific disciplines, where the rate of data acquisition and increasing modeling complexity is far outpacing improvements in processor speed. Even the amount of parallelism on homogeneous computing environments is reaching massive scales, with supercomputers utilizing hundreds of thousands of processors and graphical processing units approaching thousands of parallel threads.

However, effectively utilizing these computing environments for scientific modeling involves significant challenges. While traditional optimization methods for scientific modeling are more applicable to homogeneous and highly reliable computing environments, they cannot be applied to heterogeneous and fault prone environments due to their sequential and iterative nature. Lost or invalid results must be recalculated before the optimization methods can progress, which drastically reduces performance and efficiency. Additionally, these optimization methods are based on evolving populations, which limits their scalability to the population size and prevents them from being able to utilize massive-scale computing environments.

This thesis examines modifying traditional global optimization methods to address the challenges in massive-scale computing. Asynchronous versions of differential evolution, genetic search and particle swarm optimization were implemented for use on massive-scale computing environments. A generic asynchronous, or non-iterative, strategy for global optimization is applied to all three search types, resulting in variants that can easily scale to hundreds of thousands of computing hosts, that are automatically load balanced by design, and are unaffected by unresponsive

hosts. Additionally, verification strategies are examined which prevent malicious or faulty hosts from affecting the progress of the optimization.

The asynchronous optimization methods have also been tested using simulated environments and challenging benchmark equations to measure the scalability and the effect of heterogeneity on these methods. When compared on simulated homogeneous computing environments, representative of supercomputers or graphical processing units, traditional synchronous methods are shown to generally be unable to achieve massive scales, while the asynchronous methods are shown to improve linearly as more hosts are added. In addition, the asynchronous optimization methods are shown to be resilient to highly heterogeneous computing environments, with results showing that increasing the heterogeneity of the time it takes for results to be calculated for an optimization method either has no effect or can even improve the number of results required to reach a solution.

In addition to simulated environments, the asynchronous optimization methods have been tested on different actual computing environments using a representative astroinformatics application which evaluates the fitness of three dimensional models of the Milky Way galaxy. Optimization has been done on RPI's CCNI BlueGene/L supercomputer as well as the MilkyWay@Home volunteer computing system. MilkyWay@Home currently consists of over 25,000 active computing hosts from over 15,000 volunteers, providing a highly heterogeneous and fault prone computing environment, and this thesis has enabled the use of this powerful computing system (currently running at 516 teraflops) to advance knowledge of the structure, origin and evolution of the Milky Way galaxy. In order to perform effective optimization using the MilkyWay@Home volunteer computing project, an efficient validation strategy was also implemented to prevent incorrect or malicious results from affecting the optimization.

As computing environments continue to become increasingly distributed and heterogeneous, even at the processor level, this thesis shows that optimization methods must embrace an asynchronous and distributed computing ideology to remain effective and efficient. The results show that the generic asynchronous optimization strategy developed provides a strong starting point for the development of even more

efficient and highly scalable distributed optimization methods.

This thesis continues as follows. Chapter 2 discusses related work in global optimization methods, with background material on different local optimization methods. Various approaches for distributing these methods are also discussed. Chapter 3 describes different massive scale computing systems and gives details on the ones used in this work. Chapter 4 presents the approach and implementation of a framework for generic distributed optimization (FGDO). A general strategy for generic asynchronous optimization is given in Chapter 5 and how this was used to develop asynchronous versions of differential evolution, genetic search and particle swarm evolution. Chapter 6 describes the extensions to FGDO to allow simulation of different computing environments and describes the different benchmark optimization problems used for this. A technique for reducing the amount of verification required for using asynchronous optimization on actual volunteer computing systems is presented in Chapter 7. Results are shown examining the synchronous and asynchronous optimization methods on simulated and actual environments in Chapter 8. Future work and concluding remarks are given in Chapters 9 and 10, respectively.

CHAPTER 2
Global Optimization Methods

Global optimization methods have been widely researched. In general, they are all variations of Monte Carlo search. These approaches can be divided depending on the type of search space they optimize over. Genetic search (GS), particle swarm optimization (PSO) and differential evolution (DE) typically operate over a continuous search space, while simulated annealing (SA) and tabu search (TS) operate over a discrete (or non-continuous) search space. However, all approaches have been modified or hybridized for use in either search space. Hybridization is also a very popular strategy for enhancing the performance of these global search methods, because while they provide effective methods for exploration, or finding new potential areas for the global minimum, they suffer in exploitation, or the ability to quickly converge to a minimum within one of these areas. So while hybridizing GS, PSO and DE with SA or TA is an effective method to make one type of global search continuous or discrete, hybridizing these searches with efficient local search methods such as conjugate gradient descent (CGD), the Nelder-Mead Simplex method and the proximal bundle method (PBM) proves an effective way to search with both strong exploration and exploitation capabilities.

The remainder of this chapter proceeds by discussing the different global optimization methods. Differential evolution (DE), genetic search (GS), particle swarm optimization (PSO), simulated annealing (SA) and tabu search (TS) are discussed in Sections 2.1, 2.2, 2.3, 2.4, and 2.5, respectively. In particular, for each section the global search method is first introduced, then hybridization strategies are examined, and lastly different parallelization approaches are discussed.

2.1 Differential Evolution

Differential evolution is an evolutionary algorithm used for continuous search spaces developed by Storn and Price over 1994–1995 [84]. Unlike other evolutionary algorithms, it does not use a binary encoding strategy or a probability density func-

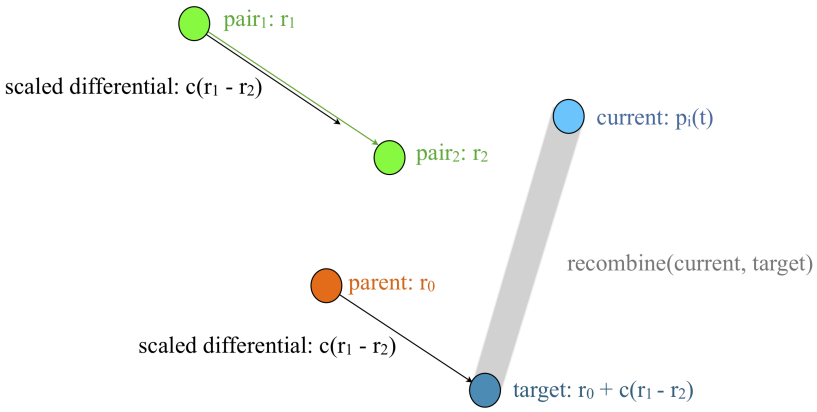

Figure 2.1: A two dimensional example of how an individual moves in differential evolution.

tion to adapt its parameters, instead it performs mutations based on the distribution of its population [77]. For a wide range of benchmark functions, it has shown to outperform or be competitive with other evolutionary algorithms and particle swarm optimization [92].

In differential evolution, an initial population is generated randomly, and each proceeding iteration generates the members of the next population by selecting a parent individual and modifying this parent using a difference vector calculated by a set number of other individuals and a recombination operator (Figure 2.1 gives an example of how an individual moves in two dimensions). The individuals improve monotonically. If the newly generated individual is more fit than the previous individual at its position in the population, the newly generated individual replaces the previous one, otherwise the new individual is discarded. Differential evolution uses the following naming strategy: *DE/x/y/z*, where *DE* simply means differential evolution, x indicates how the parent is chosen, y is the number of pairs of individuals chosen to modify that parent and z is the recombination operator (for the rest of this thesis the *DE/* is omitted).

Mezura-Montes *et al.* study different variants of differential evolution on a broad range of test functions [62]. In general, a new individual $x(l+1)$ for a new population $l+1$ is generated from the individual $x(l)$ from the previous population l. The j^{th} parameter is calculated given p pairs of random individuals from the

population l, where $r(l)^0 \neq ... \neq r(l)^{2p}$. θ, ϕ and σ are the user defined *parent scaling factor*, *recombination scaling factor* and *crossover rate*, respectively. $f(x)$ is the fitness of individual x. D is the number of parameters in the optimization function. $b(l)$ is the best individual in the population l. The different variants used are as follows:

- ***best/p/bin*** selects the best member of the population, adds the sum of the differential between p other pairs of distinct individuals and performs binomial recombination with an individual to generate its child as follows:

$$x(l+1)_j = \begin{cases} b(l)_j^0 + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] & \text{if } r(0,1) < \sigma \text{ or } j = r(0,D) \\ x(l)_j & \text{otherwise} \end{cases} \tag{2.1}$$

  Binomial recombination combines the current individual with the parameters generated by the parent and the pairs by selecting at least one parameter randomly and performing a weighted average using the recombination scaling factor.

- ***rand/p/bin*** is identical to *best/p/bin*, however instead of using the best individual as a parent, it selects a random individual (different from the random individuals used to form the pairs):

$$x(l+1)_j = \begin{cases} r(l)_j^0 + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] & \text{if } r(0,1) < \sigma \text{ or } j = r(0,D) \\ x(l)_j & \text{otherwise} \end{cases} \tag{2.2}$$

- ***best/p/exp*** is identical to *best/p/bin* except it uses exponential recombination instead of binomial recombination. Instead of selecting random parameters to recombine, exponential recombination selects a random parameter and all the subsequent parameters:

$$x(l+1)_j = \begin{cases} r(l)_j^0 + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] & \text{from } r(0,1) < \sigma \text{ or } j = r(0,D) \\ x(l)_j & \text{otherwise} \end{cases} \tag{2.3}$$

- ***rand/p/exp*** is identical to *best/p/exp* except that the parent is selected randomly, as in *rand/p/bin*:

$$x(l+1)_j = \begin{cases} r(l)_j^0 + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] & \text{from } r(0,1) < \sigma \text{ or } j = r(0,D) \\ x(l)_j & \text{otherwise} \end{cases} \tag{2.4}$$

- ***current-to-best/p*** uses the best individual as a parent to perform another differential with the current individual. This and the sum of the differential of p pairs are added to the current individual without any recombination:

$$x(l+1)_j = x(l)_j + \theta[r(l)_j^0 - x(l)_j] + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] \tag{2.5}$$

- ***current-to-rand/p*** is the same as *current-to-rand/p* however it uses a random individual instead of the best individual as the parent:

$$x(l+1)_j = x(l)_j + \theta[b(l)_j^0 - x(l)_j] + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] \tag{2.6}$$

- ***current-to-rand/p/bin*** is the same as *current-to-rand/p* however it also uses binomial recombination:

$$x(l+1)_j^c = x(l)_j + \theta[r(l)_j^0 - x(l)_j] + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] \tag{2.7}$$

$$x(l+1)_j = \begin{cases} x(l+1)_j^c & \text{if } r(0,1) < \sigma \text{ or } j = r(0,D) \\ x(l)_j & \text{otherwise} \end{cases} \tag{2.8}$$

- ***rand/p/dir*** selects a random parent and adds the differentials to this value. However, the differentials are calculated by subtracting the worse individual in the pair from the better individual:

$$x(l+1)_j = r(l)_j^1 + \phi \sum_{k=1}^{p} [r(l)_j^{1k} - r(l)_j^{2k}] \text{ where } f(r(l)^{1k}) < f(r(l)^{2k}) \quad (2.9)$$

The variants were tested using unimodal and separable problems, unimodal and non-separable problems, multi-modal and separable problems, and multi-modal and non-separable problems. The variants *rand/1/bin*, *best/1/bin*, *current-to-rand /1/bin* and *rand/2/dir* were shown to be the best for unimodal and non-separable problems, and *best/1/bin*, *rand/1/bin* and *rand/2/dir* provided the best results for unimodal/non-separable and multi-modal/separable problems. For the hardest problem set, multi-modal and non-separable, *rand/2/dir* performed the best, followed by *rand/1/bin*, with the third best results by *best/1/bin* and *current-to-rand/1/bin*. However for the hardest test function, the generalized Rosenbrock's function, *rand/1/exp* had the best performance. It is interesting to note that *best/1/bin* performed extremely well for almost all test problems, however the best performing variant typically varied as the test functions changed.

Mezura-Montes *et al.* also have examined a modification of *rand/1/bin* for constrained optimization [61]. Their approach allows the generation of multiple offspring, as well as using both the current parent and the best known solution (similar to PSO) in the generation of offspring. Another modification is that instead of only using the fitness of the children to determine if the current member of the population is updated, multiple selection criteria are used. The following calculation is used to generate a child c for member $x(l)$ of population l:

$$c_j = r(l)_j^0 + \theta^1[b(l)_j - r(l)_j^1] + \theta^2[x(l)_j - r(l)_j^2] \quad (2.10)$$

Where θ^1 and θ^2 are parental scaling factors, and $r^0 \neq r^1 \neq r^2$ are different random individuals from population l. The following selection criteria are used to determine if a child or parent is selected for the next generation:

- If both solutions are feasible, the individual highest fitness is selected.
- A feasible individual is selected over an infeasible individual.
- If both individuals are infeasible, the individual with the lowest sum of constraint violation is selected.

In addition to this, there is a user defined chance to only use the fitness of an individual as the selection criteria – in this way, infeasible solutions in promising areas of the search can remain in the population. For a wide range of test functions, this approach is shown to competitively converge to the optimum solution.

Rahnamayan *et al.* have proposed opposition-based differential evolution, which in addition to evaluating the next population, generates an *opposite population*, and uses the best value of the two to update the current population [73]. The opposite of a population P, OP can be calculated using the minimum and maximum parameter values of each member of P ($P_{min,j}$, $P_{max,j}$):

$$OP_{k,j} = P_{min,j} + P_{max,j} - P_{k,j} \tag{2.11}$$

The effect of generating an opposite population was tested for both the initial randomly generated population, and the proceeding generations generated by differential evolution. For a set of seven test functions, the initial random population had an average fitness improvement from 4% to 70% by selecting the most fit member between the original and opposite populations. By using a population and its opposite, the convergence rates improved significantly, requiring 42% to 86% less function evaluations.

2.1.1 Hybrid Differential Evolution

Zhang and Cai have hybridized differential evolution with orthogonal design, using it as a recombination operator in conjunction with *rand/1/exp* [101].

Gong *et al.* have hybridized differential evolution with particle swarm optimization [44]. Their approach is to first perform an iteration of particle swarm optimization, then perform differential evolution on the population of best found particles (for more details about particle swarm optimization, see Section 2.3). By

performing DE on the best found particles, the diversity of the particle swarm is increased and premature convergence of particles to local minima is prevented. This approach was tested with the Rosenbrock, Rastrigrin and Griewank functions, and is shown to outperform both traditional DE (*rand/1/bin*) and PSO.

2.1.2 Parallel Differential Evolution

Tasoulis *et al.* use parallel virtual machines (PVM) to implement a parallel differential evolution algorithm [88]. This algorithm generates a ring of subpopulations, and for each iteration determines which individuals a subpopulation will migrate to the next in the ring. Individuals are probabilistically selected for migration via a user selected migration constant. They test this approach with a number of test equations: Sphere, Rosenbrock, Step, Quartic, Shekel's Foxholes, Corana Parabola and Griewank. In particular, they measure the effect on convergence as the migration constant changes. Intermediate values of the migration constant result in the best convergence rates, with values close to 0 or 1 resulting in significant increases in convergence time. As found in other research, the *best/1/bin* tended to be the most efficient on across the test functions, however fine tuning the migration constant resulted in comparable or better performance for other DE strategies. This work is further expanded upon by Parsopoulos *et al.* for multi-objective optimization [66].

2.2 Genetic Search

Genetic search is a population based global optimization method. In the simplest form, an initial population is selected randomly in the search space and following this *crossover* and *mutation* are used to generate successive populations. For continuous search spaces, the most common crossover operator is simply to take two parameter sets in the population and average them. Mutation will typically take a parameter set, select a single parameter at random from within that set, and replace it with a perturbation (many select a random point within the range of possible values, and decreases the range around the point as the search progresses).

2.2.1 Hybrid Genetic Search

Many hybrid approaches have been used in genetic algorithms. Chelouah and Siarry examine a hybrid Nelder-Mead simplex and genetic search algorithm for multi-minima functions [16]. Their approach selects a wide initial population distributed among the search space in different *neighborhoods* [14]. In this method, a large number of initial parameter sets are generated, then different neighborhoods are chosen, with a center and radius. Parameter sets are then chosen such that no parameter sets share a neighborhood. Their hybrid method first performs a *diversification* step using a genetic search until stopping conditions are met and a good potential minima is found. Following this there is an *intensification* step where the best point in the genetic search is used to generate a simplex and perform a simplex search. These two steps are repeated until stopping criteria are met. This search is compared to a various range of other methods such as basic genetic search and tabu search for 21 classical test functions and is shown to be comparable or better.

Wei *et al.* propose another hybrid Nelder-Mead simplex and genetic search for multi-modal (multi-minima) functions [97]. They use three separate phases to perform the search. The first is the basic genetic search for generating a population, doing crossover and mutation. Following this, the population is divided into niches, which are determined similar to the neighborhood method. N iterations of the simplex search are performed in each niche, by choosing the best member of the niche and two other randomly chosen members. After the simplex searches have been performed in each niche, another simplex is performed by choosing the globally best member, and two other randomly selected members. These three phases are repeated until the search has completed. The hybrid search is evaluated using Schaffer's function F6, which is one of the most difficult standard test functions, and it is shown that it has the potential (with correct parameters) to converge quickly and reliably to the global optimum.

Renders and Bersini propose two methods to hybridize genetic search and hill climbing methods [74]: interleaving genetic search with hill-climbing and using hill climbing to create new crossover operators for the genetic search. The interleaving

approach performs iterations of the simplex search on each individual of the population per iteration. However, they focus on utilizing a simplex crossover operator in addition to an average crossover operator and mutation. The simplex crossover operator selects $N + 1$ members of the population and performs an iteration of the simplex search - first attempting reflection, then expanding or contracting iteratively. Both hybrids are shown to outperform the non-hybrids, with the crossover hybrid performing the best. In further work, Seront and Bersini propose a mixture of these two methods, utilizing both interleaving and a simplex crossover operator [78], for even more improved performance by gaining the benefits of both.

Hybrid genetic search/simplex has also been used with success in different fields of scientific research. Yen *et al.* [100] also use a hybrid Nelder-Mead simplex and genetic search to model metabolic systems. Their approach is to perform a concurrent simplex on the elite members of the population, in addition to traditional methods of crossover and mutation. As opposed to traditional simplex which starts with $N+1$ points, and reflects the worst point through the centroid of the remaining points, the concurrent simplex starts with $N + \Omega$ points, and reflects Ω points through the centroid of the best N. These new points are evaluated and contraction is performed as necessary. This allows multiple simplexes to be performed among the elite members of the population. Their approach is compared to adaptive simulated annealing, the simplex-GA hybrid developed by Renders-Bersini [74], two variations of genetic search and a parallel simplex method, and is shown to find better solutions to the metabolic modeling problem than the others tested.

Satapathy, Katari *et al.* evaluate genetic search/simplex hybrid and genetic search/simplex/*k-means* hybrid approaches for image clustering [75, 48]. *K-means* is a clustering method which attempts to create k clusters of vectors (in this case, parameters or individuals), with each cluster having the minimum possible sum of squares distance between its vectors. The genetic search/simplex method creates an initial population of $3N + 1$ members, and for each iteration of the population performs a simplex search with the best $N + 1$ members, and genetic search with the remaining ones. The genetic search/simplex/k-means hybrid first performs the k-means algorithm to determine one member of the initial population, generates the

rest randomly, then uses the genetic search/simplex hybrid. These approaches are shown to not be trapped in local minima, which happens with either the non-hybrid simplex or k-means algorithms. Additionally, seeding the initial population with the results of the k-means algorithm provides the fastest results.

2.2.2 Parallel Genetic Search

A wide range of parallel genetic algorithms (PGAs) have been examined for different distributed computing environments. Generally, there are three types of parallel genetic algorithms: single population (panmictic, coarse-grained), multi-population (island, medium-grained), or cellular (fine-grained) [13]. Typically, these approaches are synchronous.

Panmictic GAs create a population, evaluate it in parallel, and use the results to generate the next population. For this approach, there are three different approaches to parallelization. For a given population, each of the parameter sets in the generated population can be evaluated in parallel. This approach allows scalability up to population size used by the genetic search. Another approach is to parallelize the function evaluation and perform these iteratively for the whole population. For expensive and parallelized function evaluations, this allows as much scalability as the function evaluation can use. Lastly, a hybrid of the two approaches, parallel function evaluation done in parallel for each parameter set in the population can be done. This allows the greatest amount of scalability, but can be complicated to implement. Unfortunately, for non-parallel function evaluations or large scale computing environments none of these approaches may be able to use all the available resources.

Island approaches use multiple populations of parameter sets called *islands*. Typically, after a fixed number of iterations the populations propagate their best parameter sets to the other populations. In these cases, each island can be parallelized in the same manner as a panmictic GA, which increases scalability. Additionally, it has been shown that super-linear speedup can be attained using this method, as smaller populations can converge to minima quicker than larger populations [4, 10]. However, having populations of different sizes and/or populations running on clus-

ters of different speeds can have varying negative effects on the performance of the search.

Cellular algorithms [3, 32] evaluate individual parameter sets, then update these individual sets based on the fitness of their neighbors. Dorronsoro *et al.* have shown that asynchronous cellular GAs can perform competitively and discuss how the update rate and different population shapes affect the convergence rate [33].

P-CAGE [37] is a peer-to-peer (P2P) implementation of a hybrid multi-island genetic search built using the JXTA protocol [43] which is also designed for use over the Internet. Each individual processor (a member of the P2P network) acts as an island (a subpopulation of the whole) and evolves its subpopulation cellularly. Every few iterations, it will exchange exterior neighbors of its population with its neighbors.

There have also been different approaches taken in developing PGAs for computational grids. Imade *et al.* have studied synchronous island genetic algorithms on grid computing environments for bioinformatics [46], using the Globus Toolkit [39]. Lim *et al.* provide a framework for distributed calculation of genetic algorithms and an extended API and meta-scheduler for resource discovery [55]. Both approaches use synchronous island-style GAs. Nimrod/O [67] is a tool that provides different optimization algorithms for use with Nimrod/G [12].

Nimrod/O has been used to develop the EPSOC algorithm [53] which is is a mixture of a cellular and traditional GA. Populations are generated synchronously but the elimination of bad members and mutating good ones is done locally. Hybrid approaches [53, 82] have also been examined.

2.3 Particle Swarm Optimization

Particle swarm optimization was initially introduced by Kennedy and Eberhart [49, 34] and is a population based global optimization method based on biological swarm intelligence, such as bird flocking, fish schooling, etc. This approach consists of a population of particles, which "fly" through the search space based on their previous velocity, their individual best found position (cognitive intelligence) and the global best found position (social intelligence). The population of particles

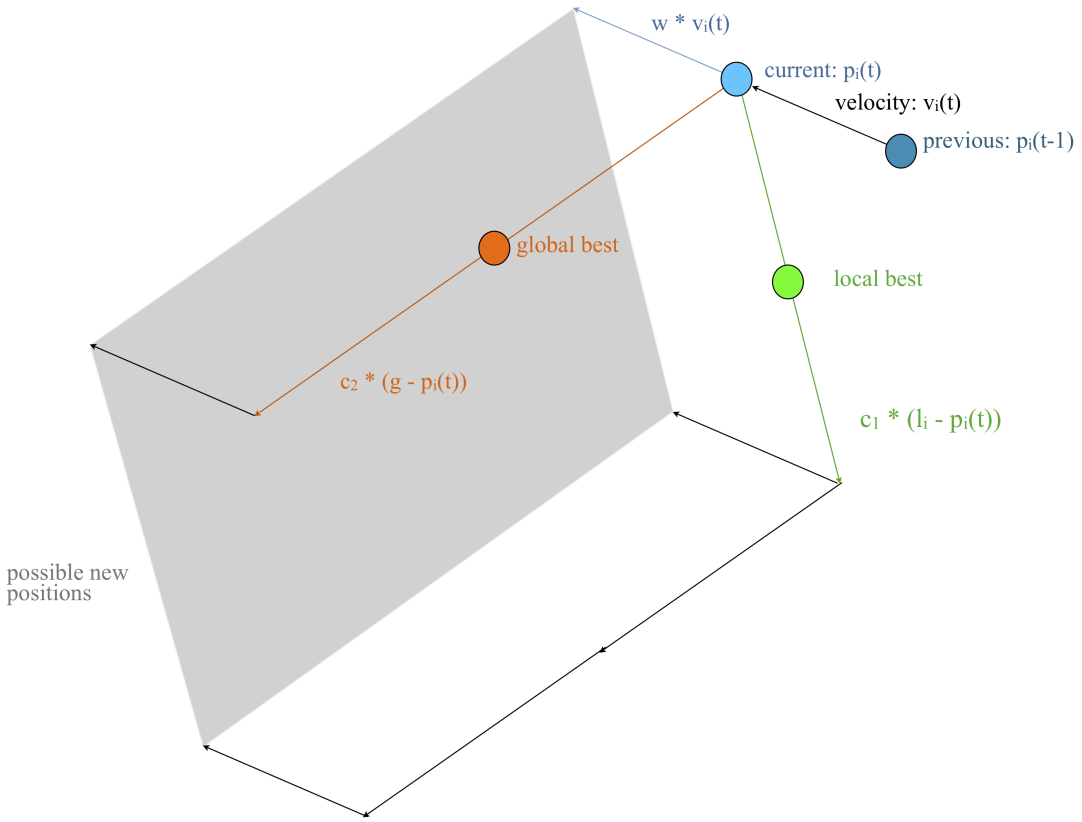

Figure 2.2: A two dimensional example of how a particle moves in particle swarm optimization.

is updated iteratively as follows, where x is the position of the particle at iteration t, v is it's velocity, p is the individual best for that particle, and g is the global best position (Figure 2.2 shows how a single particle can move in two dimensions):

$$v_i(t+1) = v_i(t) + c_1 * rand() * (p_i - x_i(t)) + c_2 * rand() * (g_i - x_i(t)) \tag{2.12}$$

$$x_i(t+1) = x_i(t) + v_i(t+1) \tag{2.13}$$

Two user defined constants, c_1 and c_2, allow modification of the balance between local (cognitive) and global (social) search. Later, an inertia weight ω was added to the method by Shi and Eberhart to balance the local and global search capability of PSO [80] and is generally used by most modern PSO implementations:

$$v_i(t+1) = \omega * v_i(t) + c_1 * rand() * (p_i - x_i(t)) + c_2 * rand() * (g_i - x_i(t)) \quad (2.14)$$

Another modification that receives wide use is that of a constriction factor as introduced by Clerc [18], but this is a special case of the inertia weight, as discussed by Eberhart and Shi [35]:

$$v_i d = K * [v_i d + c_1 * rand() * (p_{id} - x_{id}) + c_2 * rand() * (p_{gd} - x_{gd})] \quad (2.15)$$

$$K = \frac{2}{\mid 2 - \varphi - \sqrt{\varphi^2 - 4\varphi} \mid}, \varphi = c1 + c2, \varphi > 4 \quad (2.16)$$

A wide range of modifications and expansions to the PSO algorithm exist. Quantum particle swarm optimization (QPSO) is an improvement to PSO with stronger global convergence properties and is simpler to optimize because there is only one constant to be specified [86, 85]. Particles move according to a wave function, as opposed to a combination of their previous velocity, and the local and global best particles. They define new operators for the local best particle position and particle movement:

$$x(t+1) = p \pm \beta * \mid mbest - x(t) \mid * \ln(1/u) \quad (2.17)$$

$$mbest = \frac{1}{M}\Sigma_{i=1}^{M} P_i = (\frac{1}{M}\Sigma_{i=1}^{M} P_{i1}, \frac{1}{M}\Sigma_{i=1}^{M} P_{i2}, ..., \frac{1}{M}\Sigma_{i=1}^{M} P_{id}) \quad (2.18)$$

$$p_{id} = \varphi * p_{id} + (1 - \varphi) * p_{gd}, \varphi = rand() \quad (2.19)$$

Feng *et al.* have used QPSO for digital FIR filter design with better performance than normal genetic and particle swarm searches [36].

Liu *et al.* describe a version of QPSO that applies a mutation operator to each particle which improves convergence to the global minima [57]. In addition to the new quantum operators for determining the next state of a particle, mutation is applied to each particle to prevent premature convergence to local minima by mutating the particles position using a Cauchy distributed random value ($f(x) =$

$\frac{a}{\pi*(x^2+a^2)}$), with expected value 0. This distribution was chosen over a Gaussian distribution because it is able to make larger perturbations. Their results show that for the three test functions used (Rastrigrin, Rosenbrock and Griewank) QPSO without mutation provides an improvement over basic PSO, and that QPSO with mutation improves convergence to the global best solution.

Liu and Sun also describe a version of QPSO that applies an operator based on immune memory and vaccination [56]. Their approach is to generate M particles according to QPSO with mutation, as described above, and an additional N particles randomly. Following this, M particles are selected according to an antibody probability distribution function:

$$p_s(x_i) = \frac{\Sigma_{j=1}^{N} \mid f(x_i) - f(x_j) \mid}{\Sigma_{i=1}^{M}\Sigma_{j=1}^{N} \mid f(x_i) - f(x_j) \mid} \tag{2.20}$$

An additional R particles are *vaccinated*, *i.e.* they are selected randomly from the remaining particles, and replaced with the best found R particles. This approach also provides faster convergence and a higher chance of reaching the global optimum than GPSO for the Rastrigrin, Rosenbrock and Griewank test functions.

PSO has also been extended with adaptivity. Dingxue *et al.* have examined dynamically changing the inertia weight to balance the exploration and exploitation trade-off, reducing premature convergence to local minima and improving global convergence speed [31]. They use the following to dynamically update the inertia weight, with a measure of population diversity F, for any iteration t, with constants a and b, the inertia weight w is:

$$w_t = \frac{1}{a - bF_{t-1}} \tag{2.21}$$

The population diversity F is calculated as the average distance between a particles current position and individual best position, scaled between 0 and 1. For the test functions used (Sphere, Rosenbrock, and Rastrigrin), The adaptive PSO is shown to converge more quickly to a better result than non-adaptive PSO.

Liang *et al.* have proposed three new learning strategies for PSO to ensure swarm diversity, reducing the chance of convergence to local minima [54]. These

strategies focus on choosing other positions to move particles by, as opposed to the individual and global best. The three new methods chosen are *elite learning PSO* (ELPSO), *multi-exemplars learning PSO* (MLPSO), and *comprehensive learning PSO* (CLPSO). Each of these methods generates a new position for each particle to be drawn to in each iteration as follows. ELPSO randomly choses m parameters from the global best, and the rest from the particles individual best. MLPSO choses dimensions for this particle randomly from the individual best positions of all particles. CLPSO is based on an analysis of the previous two methods. In this version m dimensions from the global best, the remaining dimensions are chosen randomly from the individual best positions of the other particles. These new methods show increased robustness in finding the global optima when compared to standard PSO.

Van Den Bergh and Engelbrecht present a variant which uses a hybrid of PSO and cooperation between multiple swarm populations, or cooperative particle swarm optimization (CPSO) [9]. Their approach is based on cooperation between multiple populations in genetic algorithms, proposed by Potter and DeJong [70]. Instead of having one swarm optimizing an n dimensional vector, there are n swarms optimizing one-dimensional vectors. At the end of each iteration, the global best positions of each swarm are shared with the others. Unfortunately, it is easier for this approach to become trapped in local minima. To alleviate this problem, their approach performs one iteration of n swarm optimization and an iteration of standard PSO with separate populations. First, the n swarm optimization is done, following this, a vector of the best solutions found is used to overwrite a random particle in the standard PSO population. Following this, the best particle is chosen, and the values are used to overwrite the value of a random particle in the corresponding n population swarm. Their approach is tested for the Rosenbrock, Griewank, Rastrigrin, Quadrick and Ackley functions. Their results find CPSO to converge in the least amount of iterations, followed by standard PSO and cooperative GA based algorithms, and that their hybrid method is very robust in reaching the global minima, which improves as more particles are used. However, the improved robustness comes at the cost of a greater time to convergence.

2.3.1 Hybrid Particle Swarm Optimization

Juang proposes a hybrid genetic search/particle swarm method to automate the design of recurrent neural/fuzzy networks [47]. Their approach generates the initial population randomly. Following this, the population is sorted by the fitness of it's members. For each iteration, the next population is generated using crossover and mutation. After this population has been sorted, the best half is marked as elite and used for particle swarm optimization. The velocities for these elite members are set to 0 if the member was generated by a crossover operator (*i.e.*, it remains unchanged), or based off the member's previous position if it was generated by mutation or particle swarm. For both types of recurrent neural network optimization tested, PSO was shown to outperform GS, while the hybrid performed the best.

Koduru *et al.* test a hybrid particle swarm/simplex approach on several benchmark problems and to fitting a gene model with observed data [51]. They define two types of global search/simplex hybrid, *tandem*, where in each iteration part of the population performs the global search and the other part performs the simplex algorithm, and *cascade*, where in each iteration the entire population performs an iteration of the global search, then the simplex method on the results. Their approach uses the *cascade* method, where for each iteration the particles are first updated as per particle swarm. Following this, the resulting particles are clustered using the k-means algorithm, and the simplex method is applied in each cluster. Their results show that applying the k-means algorithm to cluster the particles improves the effectiveness of the hybrid search, where in most cases using random clusters results in poor performance. Only one test function performed better using non-hybrid particle swarm, while in all other cases the hybrid method had the best performance.

Das *et al.* expand on this work by adding gradient information to the particle swarm calculations and compare it to the hybrid particle/swarm simplex approach [23]. With particle swarm/gradient information hybrid, for each iteration, k-means is used to cluster the particles, and within each cluster new positions are calculated in one of three ways: *best-centroid*, *centroid-worst*, and *best-worst*. Best-centroid uses the distance between the best particle and centroid of the cluster,

multiplies it by a random value and constant and adds this to the global best and local best distance as in normal particle swarm. This value is then scaled down by another constant. Centroid-worst and best-worst do the same thing, except using the distance between the centroid and worst, and best and worst, respectively. These approaches are compared in estimating the parameters of a gene network model and show that this approach reduces the number of evaluations required to reach the global optimum. Additionally, the best-centroid method is shown to perform the best of the three.

Petalas *et al.* describe an entropy-based memetic particle swarm search to compute periodic orbits of nonlinear mappings [69]. Instead of performing the local search after a specified number of iterations, as done in other approaches, they use Shannon's information entropy (SIE) [79] in order to detect that the improvement rate of the swarm has degenerated. After this occurs, local search is performed on the best position of each particle. SIE is a measure of how diverse the population is, the higher the entropy, the more diverse the population is. It is calculated as the following, where b_i is the best found fitness of particle i:

$$SIE = -\Sigma_{i=1}^{N} p_i * \log p_i \tag{2.22}$$

$$p_i = \frac{b_i}{\Sigma_{j=1}^{N} b_j} \tag{2.23}$$

When change in entropy for a certain number of iterations is lower than the threshold, then the Solis and Wets algorithm [83] is performed for local search. This approach was found to be more robust than PSO, with less chance of falling into local minima, and could also find the global minimum more efficiently.

Wang *et al.* utilize a hybrid of simulated annealing and particle swarm optimization to optimize the energy consumption of wireless sensor networks [94]. They take a cascade approach, as defined by Koduru *et al.* [51]. First, an iteration of particle swarm optimization is performed and following this, the best point in the particle swarm has simulated annealing applied to it (for more details about simulated annealing, see Section 2.4). The process of PSO then simulated annealing is repeated until stopping conditions are met. The results show that this approach pro-

vides solutions with significant energy conservation. Ge *et al.* also use this approach for Job Shop Scheduling [42].

2.3.2 Parallel Particle Swarm Optimization

Recently, particle swarm optimization has also been used in different parallel environments. Baskar *et al.* extend Fitness-Distance-Ratio PSO (FDR-PSO) for concurrent execution [8, 7]. FDR-PSO is introduced and analyzed by Peram, Veeramachaneni *et al.* [68, 90] and is a modification of PSO that only selects a single other particle to modify a current particle's direction. The particle chosen is the one with the highest fitness-distance-ratio FDR between a particles current value and another particles individual best:

$$FDR = \frac{f(p_j) - f(x_i)}{\mid p_j - x_i \mid} \tag{2.24}$$

Peram, Veeramachaneni *et al.* show FDR-PSO to perform competitively with regular PSO without requiring social or cognitive terms. Baskar *et al.*'s approach makes this concurrent by utilizing two concurrent swarm populations, one using regular PSO and the other using FDR-PSO. At the end of each iteration, the global best particle of each population is shared between the two groups. Their results using optimizing reconfigurable phase differentiated antenna arrays show that FDR-PSO and their concurrent PSO perform better than regular PSO and genetic search.

Schutte *et al.* examine parallel synchronous particle swarm for load balanced analytical test problems and load-imbalanced biomedical system identification [76]. This method evaluates each particle in parallel for each iteration. Their results show that the parallel synchronous particle swarm performs well for the load balanced test problems with near linear improvement, however for the load-imbalanced problems, performance degrades with the parallel version due to each iteration waiting for the slowest computation. From these results, they suggest that an asynchronous parallel PSO would be valuable.

Koh *et al.* implement a parallel asynchronous PSO for heterogeneous networks [52] using the Message Passing Interface (MPI) [60]. The algorithm uses an approach similar to CILK's work stealing [11], where the master processor contains

a queue of currently unevaluated particles and slave processors request particles to evaluate from the master. In this way the search proceeds similar to synchronous particle swarm, as when the result for a particle is reported to a master, the next position for that particle is generated and added to the queue of work. This approach ensures that each particle performs close to the same number of evaluations. The asynchronous method is shown to achieve nearly identical results to synchronous parallel PSO for homogeneous clusters, however on their test heterogeneous cluster of 20 processors, the asynchronous version had a clear advantage in performance, reducing computation time by 3.5. Venter *et al.* use a similar asynchronous parallel PSO and analyze it for the design optimization of a typical transport aircraft wing with similar results [91].

Cui and Potok propose a distributed particle swarm optimizer that can find a solution which may be moving in a noisy search space [21]. Their approach works as a normal particle swarm, except that every time the new position of a particle is found to be worse than the local best, the fitness of the local best particle is degraded. In this way, if a particle continuously reaches bad new positions, there is a greater chance that the environment has changed and it will start performing a wider search. This is compared to other approaches which reset the particle swarm fitness every few iterations, and those that use *sentry* particles to detect when to reset the swarm fitness. Their approach not only provides more accurate results as the environment changes, but is more suitable to distributed particle swarm because the only information it requires broadcasting is the global best particle.

Xu and Zhang use a master-slave model for parallel particle swarm for attribute reduction [98]. Their method is asynchronous, with each processor being assigned a particle, and reporting information on new global best positions to the master. The master will broadcast new global best particles to all the slaves when they are found. As with other asynchronous approaches, their results show an improvement in convergence rates, and effective convergence to global minima.

Prez and Basterrechea consider both global and local swarm topologies with asynchronous and synchronous update for parallel PSO [72]. The asynchronous PSO used performs as the method described by Koh and synchronous PSO as described

by Schutte. For the global swarm topology, particles are drawn to the global best point, which is updated after every particle evaluation in the asynchronous version, and between iterations for the synchronous version. The local swarm topology uses a local best (instead of a global best) which is the best of a set of N neighbors, typically 15% of the swarm size. This work shows that for their example problems of array synthesis and planar near-field antenna measurements, asynchronous global PSO clearly outperforms the other versions in terms of time to convergence with an additional benefit of better utilization of heterogeneous resources, however it should be noted that the local best approach has a wider search area and is more resistant to convergence to local minima.

2.4 Simulated Annealing

Simulated annealing is an optimization strategy typically used when parameters are discrete. It starts at a current point, modifies it and moves to the new point if the fitness is within a certain *temperature*. As the search progresses, it begins taking close to random steps and eventually it reaches a point where the current point will only take a step that improves its fitness.

Yeh and Fu present a parallel adaptive simulated annealing algorithm and use it for locating the activation area of functional magnetic resonance images (MRI) [99]. They use an island approach, where each processor has a certain set of solutions, and generates neighbors for these iteratively either with the classical or adaptive simulated annealing approaches. After n neighbor evaluations, individuals are selected with probability P_m and migrated to the neighboring islands. Their results show that while the parallel approach can produce good results, the scalability is low - only a factor of 2.63 for adaptive simulated annealing and 2.18 for classical simulated annealing on a cluster of four processors.

Da and Xiurun utilize a hybrid particle swarm and simulated annealing to optimize the weights of an artificial neural network for rock engineering [22]. Instead of only updating the global best particle when a new global best is found, whenever a new local best is found it replaces the best particle as in simulated annealing, *i.e.*, if $p_{lb} < p_{gb}$ otherwise with the probability $1 - e^{-\frac{p_{gb} - p_{lb}}{temp}}$. For their application, this is

shown to converge to better results than traditional particle swarm, which converges quicker to local minima.

Miettinen *et al.* examine different hybrid simulated annealing methods for global continuous optimization [63]. These methods are based on the proximal bundle method [50]. Using simulated annealing within the proximal bundle method (*i.e.*, *biased proximal bundle method*) by having the Metropolis criterion determine the serious step of the proximal bundle was shown to be inadequate. Their first hybrid method does the opposite with the proximal bundle method being used within simulated annealing by applying the proximal bundle method on candidates accepted with the Metropolis criterion by simulated annealing. The second method combines the first method with the biased proximal bundle method (BPBM), using this instead of the proximal bundle method as in the first method, with the simulated annealing and BPBM each having their own cooling schedules. The final method speeds up the first method by performing the proximal bundle method with low accuracy, and increasing the accuracy of the stopping conditions as the search progresses. These methods were compared to simulated annealing with two different parameters used for temperature reduction $N_t = max(100, 5n)$ and $N_t = 5$. Both the first and third methods were shown to be the most reliable, with the third method being the most efficient. All the hybrids outperformed simulated annealing in both efficiency and reliability.

Wang *et al.* use a parallel genetic search/simulated annealing hybrid for multi-pass milling to optimize cutting parameters [95, 96]. The hybrid method proposed combines simulated annealing into the crossover operator. Two parents generate two children, which are the best and worst members found by a Markov chain. This chain is generated with simulated annealing, with a member being selected to update the best/worst found if the Metropolis criterion is valid, *i.e.*, $min(1, e^{\frac{f_i - f_{worst}}{T_t}}) \geq r$, where f_i is the current member, f_{worst} is the worst in the chain, T_t is the temperature at time t and r is a uniformly distributed random number between 0 and 1. Their approach to parallelization is hierarchical, with a master node controlling sub-groups. There is no migration between sub-groups, however there is migration within nodes of a sub-group. A sub-group consists of eight processors, connected

using a ladder neighborhood method. After a certain number of iterations, the best 1% of each population is migrated to the processors' neighbors and replaces the worst 1%. A sub-group stops searching when the best found member has changed less than a threshold in a certain number of iterations. Their results show that this approach finds better solutions faster than normal parallel genetic search, and simulated annealing.

2.5 Tabu Search

Tabu search (TS) is an iterative search method, similar to simulated annealing. TS starts from a randomly selected initial solution s, and generates a set of neighbors s' for this solution by applying previously defined *perturbations* to s. The best solution in s' becomes the new s, even if it is worse than s. The last m previously visited s are stored in a *tabu list*, and s will not move to a neighbor if it is in that list. The algorithm typically terminates after some number of iterations have been performed without any improvement. While like simulated annealing, TS has typically been used for combinatorial problems, or optimization of non-continuous variables, later work has been done to apply it to continuous variables and global optimization. Chelouah and Siarry mention distributed tabu search [15], however no current work on this has been presented to date.

Siarry and Berthiau apply TS to continuous optimization by creating different perturbation operations to generate neighbors in a continuous space [81]. Neighbors are generated in a *ball* around the current solution s. Because typically the number of neighbors created should be small, simply choosing random parameters would lead to an inhomogeneous selection of points inside that ball. To overcome this problem, they propose three different types of partitioning which points are selected within, *geometrical*, *linear*, and *isovolume*. Geometrical partitioning selects points between radii that are calculated according to a geometrical progression of ratio 2, linear partitioning selects points between equally distanced radii from s, and isovolume partitioning selects points between equally volumed radii from s. Their results find that for certain problems, continuous tabu search can be comparable to or better than simulated annealing. Additionally, geometrical and linear partitioning tended

to be better than isovolume.

Chelouah and Siarry have enhanced their continuous tabu search for global optimization [15]. This enhanced tabu continuous search (ETCS) improves upon basic continuous tabu search by first performing diversification to locate promising regions, then performing tabu search within those promising regions. Additionally, ETCS uses hyper-rectangles instead of balls, due to the ease of computing neighbors. Diversification is done by finding *promising areas*. A new promising area is found when a neighbor is worse than the current solution by a certain threshold and is not in the ball of an already found promising area. After a certain number of iterations for finding promising areas, the best promising area is chosen by first removing all promising areas with fitness worse than the average fitness of the promising area solutions. Following this, the hyper-rectangular balls around each promising solution are halved in size, and the best neighbor of each is chosen if it is better than the current promising solution. The worst new promising area solution is removed. It iteratively repeats the reduction in hyper-rectangle size, and removal of the worst promising area until one remains. Finally it performs the regular continuous tabu search on the remaining promising area. Results show that ETCS performs better than other versions of continuous TS for functions of variable length 10 or less, and competitively for those of larger size.

Chelouah and Siarry later hybridize ETCS with the Nelder-Mead simplex method [17]. This approach first performs diversification the same as ETCS, however upon finding the best promising area, a simplex search is performed from that point. After the local simplex search is completed, the search again performs diversification from this new point. This process repeats until stopping conditions are met: a maximum number of iterations have been performed or a number of iterations have been performed without improvement. The hybrid is shown to improve upon the results of ETCS, and outperform basic simulated annealing and genetic search for problems with small amounts of variables (less than 5).

Wang *et al.* improve upon Chelouah and Siarry's ETCS approach by allowing the evaluation of additional neighbors within the central most hyperrectangle [93], something which was avoided in their work to enforce the tabu search moving signifi-

cantly away from the current solution to avoid convergence to local minima [81, 15]. For the test functions tested, it is shown that the intensification of the search in the interior hyperrectangle improves the rate to solution by nearly half, without reducing accuracy.

Franz and Speciale also apply tabu search to global optimization of continuous problems [40]. They use a dynamically sized grid to determine the neighbors of a solution s, and refine the grid as the search progresses. To reduce the number of evaluations performed, a neighbor is chosen by evaluating neighbors above and below the solution for each variable in the function, which requires only $2n$ evaluations at most, instead of 2^n evaluations if all possible neighbors were evaluated. Additionally, the first neighbor that improves upon the current solution is chosen, if one is found, and after this the next neighbor chosen to evaluate is in the same direction as the previously selected neighbor. The grid size is reduced when no good neighbors are found. This makes the algorithm converge to local minima. To escape these, when the grid size is reduced to a certain threshold, the search will then move to the best neighbor, even if it is worse, as opposed to the first found better neighbor. The previous minimum found as well as the previous neighbors are kept in tabu lists so they will not be selected again. The algorithm will restart itself from a random starting point if it falls into the basin of a previously found minimum and cannot improve. This continues until the algorithm has restarted itself a certain number of times. Their approach is shown to provide good solutions for device parameter extraction and a suite of test functions, including the Dejoung, Goldstein, Hartman, Rosenbrock, Shubert, and Zakharov functions.

CHAPTER 3
Massive Scale Computing Systems

Massive scale computing systems are becoming more common as researchers gain access to new computational resources and share existing ones. Volunteer computing grids are also gaining popularity because they provide access to potentially millions of computing hosts. As these systems reach larger and larger scales, they are becoming extremely powerful yet increasingly challenging to utilize effectively and efficiently. These computing systems also come with varying costs, heterogeneity, communication speeds and volatility. This chapter examines the three main types of massive computing systems: supercomputers in Section 3.1, computational grids in Section 3.2, and lastly volunteer computing grids in Section 3.3. These sections also discuss the representative test environments for each of these used in this research.

3.1 Supercomputers

Supercomputers typically use thousands of lower speed processors connected with very low latency and high bandwidth links. Applications are typically developed using vendor specific versions of the *Message Passing Interface* (MPI) [60]. This allows applications to scale to the large number of processors due to the extremely low latency, which enables a high communication to computation ratio. The processors are also homogeneous, which makes load balancing and distribution easier. Typically resources are dedicated to a single user and batch job scheduling is used to manage multiple users submitting jobs which are queued and executed in order of priority and submission time.

Figure 3.1 shows the number of cores in the top 100 supercomputers in the world based off their computing power rank in the top 500 list [1]. The trend shows dramatically how dependent the computing power of a supercomputer is based on the number of computing cores, with the most powerful supercomputers having over 100,000 computing cores. As this trend continues, scalability will become one of the

[1] http://top500.org

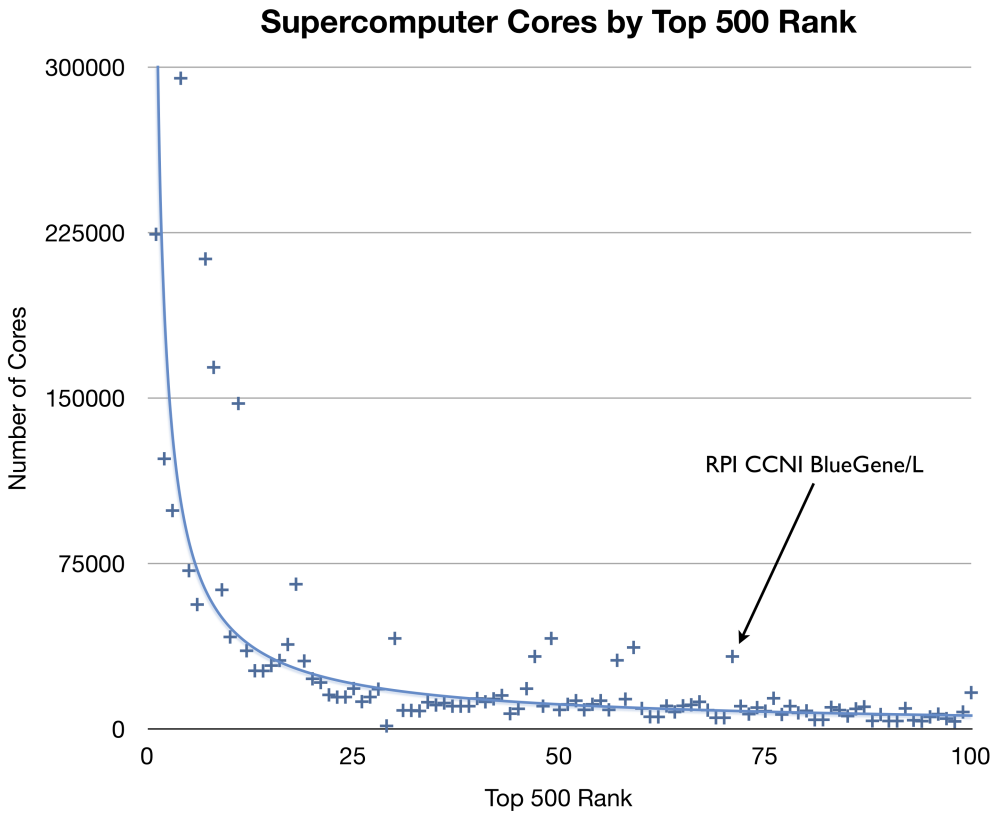

Figure 3.1: The number of computing cores in the top 100 supercomputers based on their top 500 list rank.

key issues in developing effective software for supercomputing systems.

3.1.1 CCNI BlueGene at RPI

The supercomputing environment used in this work is RPI's IBM BlueGene/L system. Our experiments used one rack of 1024 nodes, each with two 700MHz PowerPC 440 processors with 1GB RAM, for a total of 1TB RAM across the entire rack. Inter-node communication is provided by a 3-dimensional torus with 175MBps in each direction, and 1.5μusec latency. Each node can be run in non-virtual mode, with one processor performing communication and the other computation, or in virtual mode, with both processors performing computation and communication. The current system consists of 5 partitions, one 512 node partition, and four 128 node partitions. This supercomputer provides a very high performance computing

environment with which to run more traditional optimization algorithms, so they can be compared to their asynchronous versions on more heterogeneous environments.

3.2 Grid Computing

Computing grids are named after the power grid, with a similar goal in mind. The intention of grid computing environments is to allow scientists and developers to easily access globally distributed computing resources, just as people easily access distributed power sources through electrical outlets. Computational grids usually span institutions with large geographical distances, and consist of clusters of high performance processors with low latency connected by high bandwidth but high latency connections between institutions due to the distances involved. A common issue for computational grids is dealing with inter-institutional firewalls, scheduling and accounting details.

Different software packages have been developed to aid in easily using computational grids. The Globus Toolkit [38] provides a set of tools and discovery services for applications running on grids. These tools enable users to develop and deploy applications and obtain information about resource availability. While not providing an execution environment, Globus does allow grid users to schedule and deploy applications on a heterogeneous environment by providing the required information to determine what type of binaries to run and when to run them. A common management system used by Globus-enabled grids is Condor, which is a distributed resource management system that is designed to support high-throughput computing by harvesting idle resource cycles [41].

In contrast to Globus, which provides a set of tools to enable grid computing, Legion provides a virtual operating system which operates over a distributed set of host computing nodes [64, 45]. Legion supports multiple programming languages and uses a single unified object model and programming-level abstractions to hide the complexity of the underlying grid from the user and simplify the software development process. Legion also provides a web-based user interface for job scheduling, monitoring and visualization.

Another approach to grid computing is to use distributed programming lan-

guages and middleware. SALSA is a run-time system provides a distributed execution environment based on Java, which allows for execution on and communication between heterogeneous hosts [89]. Additionally, the actor model [2], transparently provides asynchronous distributed communication through message passing and migration. Malleability allows for even greater levels of dynamic reconfiguration by allowing generic reshaping of applications [27, 25]. The actor model facilitates concurrent and deadlock resistent applications, in addition to language and library based support for reconfiguration by transparent migration, and semi-transparent malleability.

The Internet Operating System (IOS) [59] is a modular middleware that enables autonomous reconfiguration of distributed applications. IOS is responsible for the profiling and reconfiguration of applications, allowing an application's computational entities to be transparently reconfigured at runtime. IOS interacts with applications through a generic profiling and reconfiguration interface, and can therefore be used with different applications and programming paradigms, such as SALSA [24] and MPI [58]. Applications implement an interface through which IOS can dynamically reconfigure the application and gather profiling information. IOS uses a peer-to-peer network of agents with pluggable communication, profiling and decision making modules, allowing it to scale to large environments and providing a mechanism to evaluate different methods for reconfiguration.

3.2.1 RPI Grid

The heterogeneous grid environment tested used a Power-PC (PPC) cluster and two Opteron (OPT) clusters. The PPC cluster consists of two single-processor, two dual-processor and four quad-processor single-core Power-PC processors running at 1.7GHz. Each processor has 2GB RAM, for a total of 44GB RAM. Intra-cluster communication uses 1GB/sec bandwidth, 100μsec latency Ethernet. The first Opteron cluster (4x2 OPT) consists of 10 dual-core quad-processor machines, and the second (4x1 OPT) consists of 4 single-core quad-processors, all running at 2.2GHz. The dual-core nodes have 32GB RAM, for a total of 128GB RAM, and the single-core nodes have 16GB RAM for another 160GB RAM (192GB in all). The

OPT processors are running GNU/Linux version 2.6 as the operating system. Communication within and between the OPT clusters is provided by 10GB/sec bandwidth, 7μsec latency Infiniband, and 1GB/sec bandwidth, 100μsec latency Ethernet. The PPC cluster and Opteron clusters are connected over RPI's wide area network forming the Rensselaer Grid testbed. The RPI Grid provides a moderately sized and moderately heterogeneous environment to test asynchronous optimization.

3.3 Volunteer Computing Grids

Volunteer computing enables people across the world to volunteer their computing resources such as processors, graphical processing units (GPUs) and hard drive space. Popular examples include SETI@Home [5], which was generalized to the Berkeley Open Infrastructure for Network Computing (BOINC) [6], IBM's World Community Grid[2] and Stanford's Folding@Home [65]. Users can even volunteer non-standard computing units such as gaming consoles like the XBOX 360 and Playstation 3. These computing frameworks not only provide thousands, or even millions, of personal computers at the low price of running a server, but also generate public interest in different scientific computing projects.

However these benefits do come at a price. The computing architectures involved can be extremely different, located world-wide with dramatically different latencies and are sporadically available at the volunteers whim. The result of this is that the computing network is highly heterogeneous in terms of computing power, architecture and latency, as well as extremely volatile as there is no guarantee that results reported by a volunteered host are correct, or when they will be returned, if ever. In addition, only client-server communication is allowed which even further limits synchronization and work sharing between volunteers.

3.3.1 BOINC

BOINC was used in this thesis work for a variety of reasons. First, BOINC currently has a large existing user base and it is easy for current users to join new projects. All a user who has already installed the BOINC client needs to do to join

[2]http://www.worldcommunitygrid.org

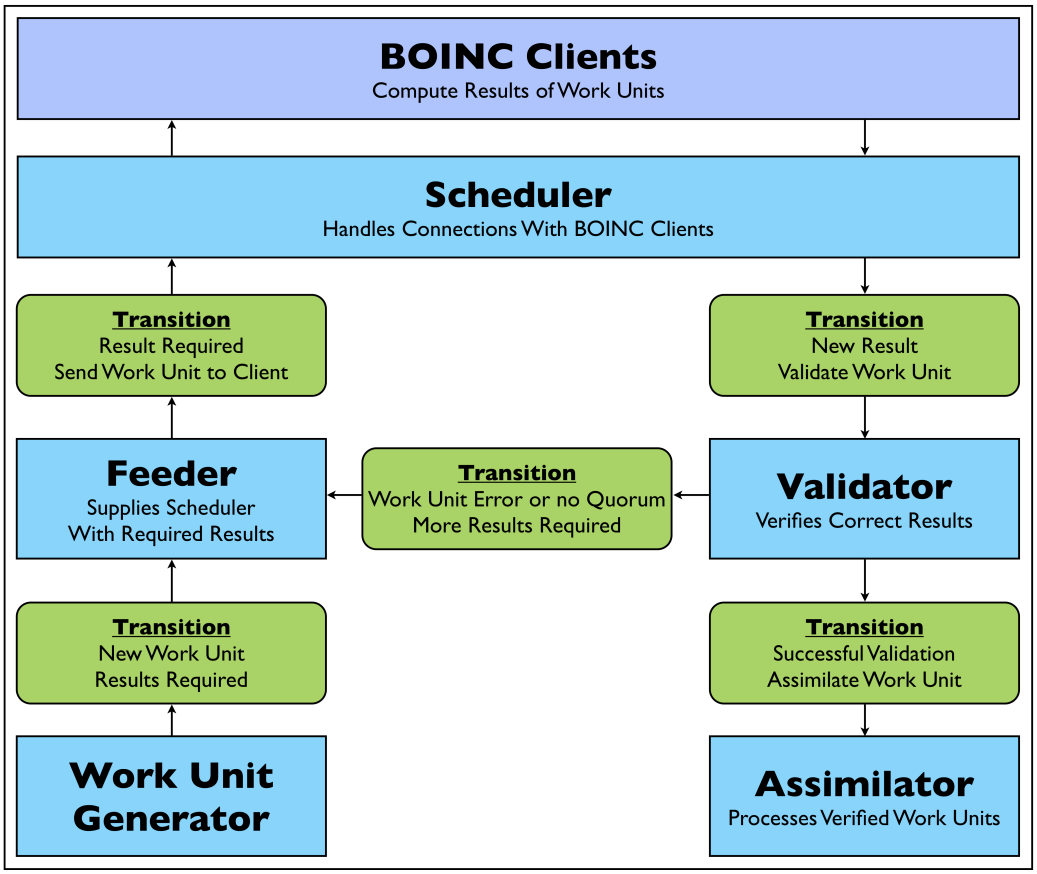

Figure 3.2: Handling of work units by the BOINC server software.

a new project is enter the project's website to join and start participating. Second, the code is open source and extensible, which allows easy modification to develop asynchronous optimization methods.

The BOINC server side software consists of six services that handle work generation, communication with clients and result verification (see Figure 3.2). Tasks generated for clients to compute are called *work units*. The six services work as follows:

- The **Transitioner** determines which work units are ready to be validated or need more results to be calculated. It will resend work to clients if a timeout has elapsed or an error occured in calculating a result. If a work unit has been assimilated, it flags that work unit as ready to have its associated files deleted.

- The **Feeder** maintains a queue of work units that are ready to be sent to

clients.

- The **Scheduler** handles incoming and outgoing communication with clients. New incoming results flag that work unit to be processed by the transitioner. This spawns CGI scripts which handle connections with clients that send work units from the ready-to-be-sent queue.

- The **Validator** is a daemon that verifies the reported results for work units. When it determines that a certain quorum of results are the same, that work unit is ready to be handled by the assimilator.

- The **Assimilator** handles work units that have been verified to be correct. After a work unit has been assimilated, it will not be sent to clients again.

- The **Work Generator** is either a daemon or a script which generates new work units to be sent to clients. After work units are generated they will eventually place them in the ready to send queue.

3.3.2 MilkyWay@Home

The Milkyway@Home project was developed using BOINC as a volunteer computing environment for this work. MilkyWay@Home currently consists of over 25,000 active hosts, with various operating systems and architectures. OS X, Linux and Windows are supported, and the client application is open source so users can compile their own versions for more obscure architectures. MilkyWay@Home also supports *General-Purpose computing for Graphical Processing Units* (GPGPU) with application versions for ATI and NVIDIA graphics cards [30]. The application's run-time is highly heterogeneous: it can take anywhere from around 20-30 seconds on a high end GPU, to over 8 hours on older CPUs. Round trip time for calculated results can be even longer, as some computers disconnect midway through calculations (which are checkpointed) and return results when they re-connect to the project. This project provides a highly heterogeneous and truly massively distributed computing environment with which to test asynchronous optimization.

CHAPTER 4
A Framework for Generic Distributed Optimization (FGDO)

This chapter describes the framework developed for generic distributed optimization (FGDO). The approach for the developement of this framework is described in Section 4.1 and implementation details are discussed in Section 4.2.

4.1 Approach

A generic distributed optimization framework should enable scientists to easily and efficiently analyze different models and hypotheses. A scientist should only need to plug in the function to be optimized with distribution details, and the framework should be able to use any appropriate search method and execution environment. The framework should be able to support the large scale computing environments available, whether supercomputers, a grid or the Internet. Likewise, fully asynchronous, partially asynchronous and fully synchronous search methods should be provided as some perform better for certain optimization problems and computational environments. The framework must support an easy interface between the search methods and distributed evaluation environment, which will allow further research into asynchronous optimization methods and distributed computing, as the search methods and distributed evaluation environments can be developed simultaneously and plugged into each other using the framework.

The different types of massive scale computing environments can require similar parallelism, albeit for different reasons. The main challenges in global optimization for supercomputing environments lies in a scalable function evaluation and effective partitioning of the problem across the available processors. If the function evaluation is nondeterministic, then effective partitioning can be especially difficult, and processing will only proceed as fast as the most expensive partition. The challenges in grid computing, apart from scalability lie in handling firewalls between clusters and the heterogeneity of the different clusters. Due to the issue

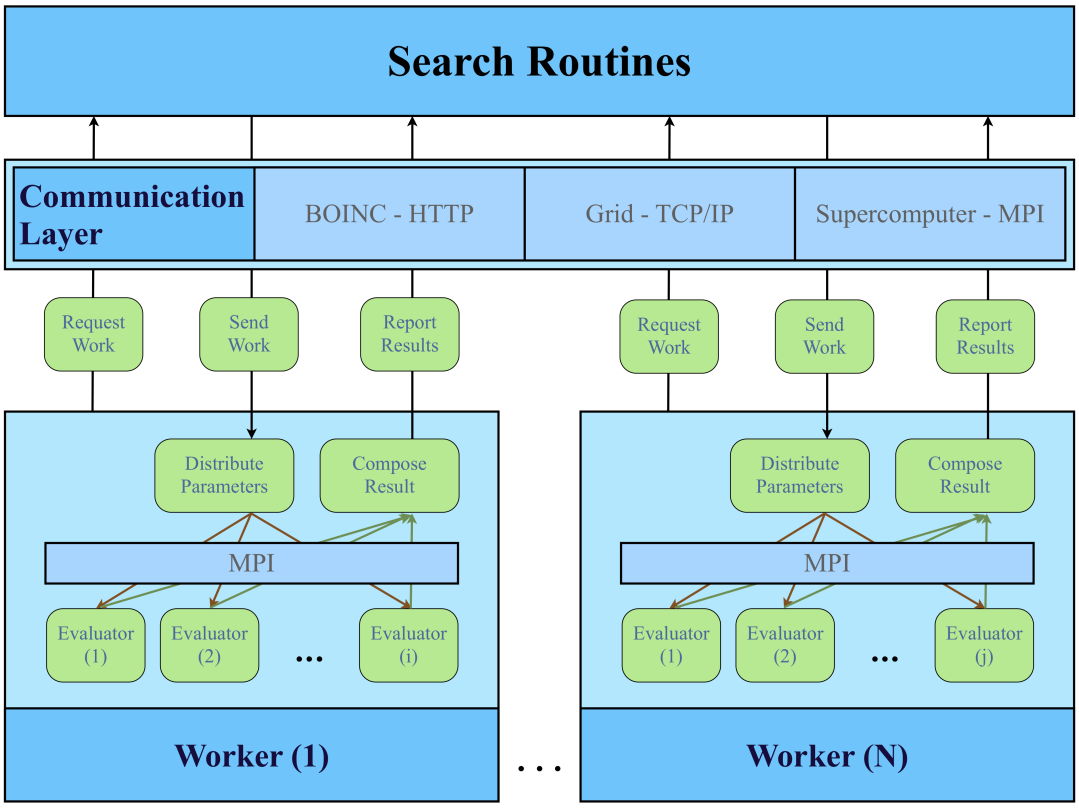

Figure 4.1: A two level parallelism strategy for computing function evaluations. Each worker can calculate the function to be optimized in parallel using MPI, compose the result and then report it to the master via the appropriate communication protocol.

of firewalls and heterogeneity of architecture and operating systems between clusters, parallelizing function evaluations over multiple clusters can be difficult and inefficient. The availability of the different clusters also presents challenges if they become unavaialble during the course of the optimization. Lastly, on internet computing frameworks, such as BOINC, parallelizing the function evaluation between different volunteer computers is highly difficult, and also requires redundancy in the case that any computer becomes unavailable. Additionally, the scalability of the optimization method is critical due to the large amount of volunteers that can be processing function evaluations concurrently.

All these issues can be addressed by a generic global optimization framework that utilizes two seperate levels of parallelism, the first in calculating multiple func-

tion evaluations in parallel and the second in parallelizing these function evaluations (see Figure 4.1). This dual layer parallelism is an extension to previous work on GMLE, a generic maximum likelihood evaluator, that unifies GMLE's previously presented asynchronous and synchronous modes [26, 87, 28, 29].

Where in the previous versions either multiple work units were evaluated asynchronously or a single function evaluation was calculated in parallel, this new distributed evaluation framework supports both which improves its scalability and efficiency. For example, on a supercomputer, this parallelism strategy allows the distributed evaluation environment to allow multiple parallelized function evaluations at maximum scalability to execute concurrently. Similarly, each cluster on a grid can perform parallel function evaluations asynchronously with other clusters, with the only inter-cluster communication coming from function evaluation requests which tend to be quite small. While an internet computing environment can have each volunteered computer process its own individual function evaluations.

Evaluations can either be pulled by workers, as done by the asynchronous search strategies discussed in Chapter 5 or pushed to workers while the search method blocks waiting for a result, as is required by synchronous search strategies.

4.2 FGDO Implementation

The FGDO implementation provides a set of three interfaces for use by developers. The first is an interface for developers of the optimization function to specify it in a way that allows it to be distributed (shown in Table 4.1). The second allows different optimization methods to easily perform synchronous or asynchronous evaluations of any optimization function developed with the optimization function interface (shown in Tables 4.2 and 4.3). The last is an interface that allows users of FGDO to easily run implemented search methods. The following sections describe these three interfaces, respectively.

4.2.1 Implementing Optimization Functions

The optimization function interface allows developers to specify how workers are created, how the optimization function is distributed amongst them, and how

Scientific Model Worker Interface		
void	read_data	(int rank, int max_rank)
void	calculate_integral	(double[] params, int n_params int rank, double[] &result, int &result_length)
void	compose_integral	(double[][] results, int n_results, int result_length, double[] &integral)
void	calculate_likelihood	(double[] params, int n_params, int rank, double[] &result, int &result_length)
void	compose_likelihood	(double[][] results, int n_results, int result_length, double &likelihood)

Table 4.1: Scientists provide the *read_data*, *calculate_likelihood*, *compose_likelihood* functions for use by workers in the distributed evaluation framework. *calculate_integral* and *compose_integral* are optional, only required if an additional integral calculation needs to be performed in parallel.

Synchronous Distributed Evaluation Framework Interface		
double	evaluate	(double[] params, int n_params, double &fitness)
double[]	evaluate_multiple	(double[][] params, int n_params, int n_sets, double[] &fitness)

Table 4.2: A search method can request evaluations to be performed synchronously by *evaluate* or *evaluate_multiple* which block until the evaluations have been performed and the fitness reported.

Asynchronous Distributed Evaluation Framework Interface		
void	enqueue	(double[] params, char[] metadata, int n_params)
void	enqueue_multiple	(double[][] params, char[][] metadata, int n_params, int n_sets)
void	result_handler	(double[] params, char[] metadata, int n_params, double fitness)

Table 4.3: Search methods can request for evaluations to be evaluated asynchronously with *enqueue* or *enqueue_multiple*. An asynchronous search method must implement a *result_handler* function which will process reported function evaluations.

the result is composed, as shown in Figure 4.1. Worker initialize with the *read_data* function, which is executed in parallel across all subprocessors using MPI if the worker is parallelized. Otherwise, individual workers on BOINC call the *read_data* and are initialized with the data passed to them by the BOINC framework.

Fitness of parameter sets is calculated using the *calculate_likelihood* and *calculate_integral* functions (if needed). If worker calculations are done in parallel, the results from each process are combined using the *compose_likelihood* and *compose_integral* functions. Workers perform work requests and report results via MPI on a supercomputer or unfirewalled grid, TCP/IP on a firewalled grid, or using the work requesting framework built into BOINC clients.

4.2.2 Implementing Optimization Methods

Optimization methods can enqueue parameter sets to be evaluated asynchronously with the *enqueue_work* or *enqueue_multiple* functions and results from these will be used to call the *result_handler* function specified by the *set_result_handler*. For grids and supercomputers, work requests can be queued, which can improve performance by having parameter sets that need evaluation be pushed to available workers as soon as they are created. On BOINC, results need to be enqueued so they can be sent to clients via the BOINC scheduling system, when clients request work.

Alternatively, optimization methods can evaluate parameter sets synchronously with the *evaluate* or multiple parameter sets in parallel, for scalable calculation on supercomputers or multiple clusters on a grid, with the *evaluate_multiple* function. Both of these function calls will block until the parameter sets specified have been evaluated.

4.2.3 Executing Distributed Optimization

A number of different synchronous and asynchronous optimization methods have been implemented using FGDO. Section 4.2.3.1 describes how to run the asynchronous optimization methods and Section 4.2.3.2 describes how to run the synchronous optimization methods.

Asynchronous Search Methods		
void	create_search	(char[] search_name, int number_arguments, char[][] arguments, int number_parameters, double[] initial_point, double[] step_size, BOUNDS* bounds)
void	read_search	(char[] search_name, void* &search_data)
void	generate_parameters	(char[] search_name, void* search_data, SEARCH_PARAMETERS* search_parameters)
void	insert_parameters	(char[] search_name, void* search_data, SEARCH_PARAMETERS* &search_parameters)

Table 4.4: Multiple asynchronous searches are managed through these four methods by the *search manager*. New parameters sets to be evaluated are generated through *generate_parameters* and results are inserted to managed searches with *insert_parameters*.

4.2.3.1 Asynchronous Optimization

Asynchronous differential evolution, genetic search and particle swarm optimization were implemented using FGDO for both MPI and BOINC. Asynchronous searches must implement the following methods, shown in Table 4.4. These are used to create an *ASYNCHRONOUS_SEARCH* struct which is used by an *Asynchronous Search Manager* to control the operation of asynchronous searches. There is an implementation of asynchronous search managers for both MPI and BOINC, and these can control multiple asynchronous searches simultaneously.

Asynchronous searches need to implement *read_search*, *create_search*, *generate_parameters* and *insert_parameters* methods. *create_search* starts a new search with the given parameters and name. *read_search* reads a checkpointed search, which allows the asynchronous search manager to keep persistent information on long running searches, which is especially important for the BOINC platform. Implemented searches are expected to checkpoint themselves when created and during the course of their optimization, so they can be restarted with the *read_search* method. *generate_parameters* requests new parameter sets to be evaluated from the search, and *insert_parameters* sends parameters and fitness values to the asynchronous search to update the population. Asynchronous search methods must also implement a *get_search* method which provides the *ASYNCHRONOUS_SEARCH*

Synchronous Optimization Methods	
synchronous_newton_method	(int number_arguments, char[][] arguments, int number_parameters, double[] initial_parameters, double[] step_size)
synchronous_gradient_descent	(int number_arguments, char[][] arguments, int number_parameters, double[] initial_parameters, double[] step_size)
synchronous_c_gradient_descent	(int number_arguments, char[][] arguments, int number_parameters, double[] initial_parameters, double[] step_size)
Synchronous Optimization Arguments	
-iterations #	Specifies the number of iterations performed.
-min_threshold #	Specifies the minimum change in fitness for the line search.

Table 4.5: This table shows the function calls for starting the synchronous optimization methods and what their different arguments are. *initial_parameters* specfies where the search starts, and *step_size* determines the step size which the gradient and hessian are calculated with.

struct that is used by asynchronous search manager. This is done using a *register_search(ASYNCHRONOUS_SEARCH search)* method. *BOUNDS* is a struct that contains the range that parameters are valid within.

4.2.3.2 Synchronous Optimization

Gradient descent, conjugate gradient descent and the newton method have been implemented using the synchronous implementation of FGDO for MPI. Table 4.5 shows how these search methods can be run and the different arguments for determining how long they run.

CHAPTER 5
Asynchronous Global Optimization

This chapter describes implementing asynchronous versions of differential evolution, genetic search and particle swarm optimization, as described in Sections 2.1, 2.2 and 2.3, respectively. Figure 5.1 presents a generic asynchronous search methodology for massive scale computing. A population of parameter sets is kept and used to generate new parameter sets which are placed in a work queue. Clients connect asynchronously and request parameter sets to evaluate from this work queue. New parameter sets are generated using different operations on the population when they are needed as the queue runs low. This can be used to perform any of the synchronous methods described in Chapter 2 so long as the results are processed

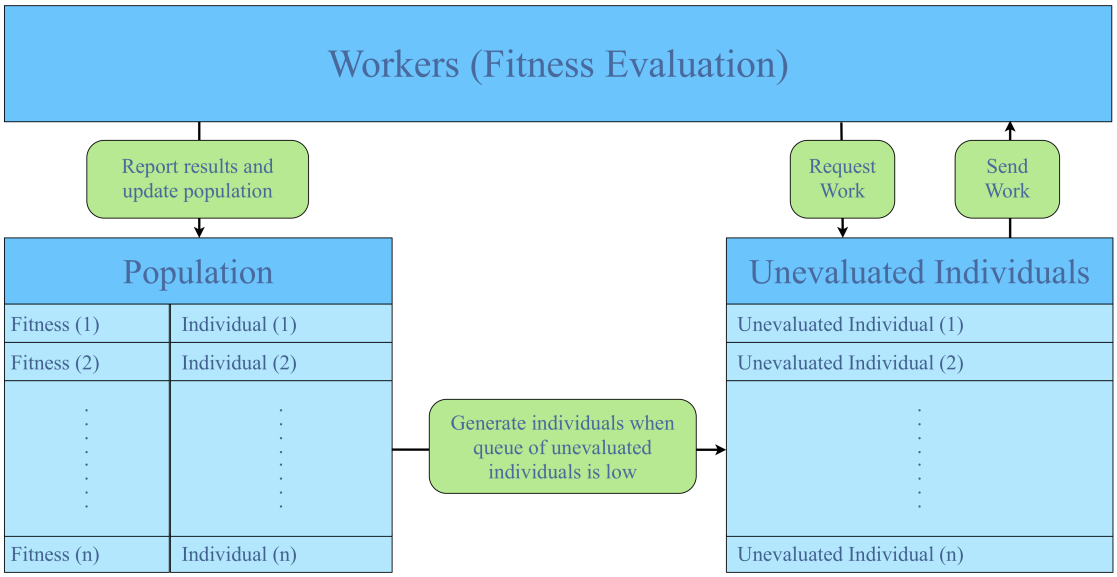

Figure 5.1: A generic asynchronous search methodology for massive scale computing systems. A population of the best known individuals and their fitness is stored and used to generate new individuals to evaluate. These unevaluated individuals are kept in a work queue which workers request work from. The work queue can request more work to be generated from the population (at any time) to ensure that work requests are answered. Evaluated individuals are used to evolve the population when their results are reported by the workers.

synchronously – they are reported in the same order that they were generated. Otherwise, the general way for extending the asynchronous search is through using different types of operators to generate new parameter sets to place on the work queue. For example, in genetic search or particle swarm optimization, the succesive generation or particles would be placed on the work queue, and when these results were reported the population would be updated.

The following sections describe modifications to differential evolution, genetic search and particle swarm optimization to enable them to perform asynchronously using the discussed strategy.

5.1 Asynchronous Genetic Search

Asynchronous genetic search can most easily model traditional genetic search and is extremely similar to steady state genetic search. The main difference is that instead of one parameter set being generated at a time, multiple are generated and evaluated asynchronously. In fact, asynchronous genetic search as performed by FGDO on the BlueGene using all processors as a single worker is no different than steady state genetic search.

Using this strategy, when new members need to be generated, different recombination operators are applied to randomly selected members of the population. When the fitness of a member is reported, it is then inserted in order into the population and the worst member of the population is then removed if the population size is greater than a fixed value. The member removed can be the member just reported. The different recombination operators used in this work were mutation, average, double shot and probablilistic simplex. Children are generated using these recombination operators as follows:

- **Mutation** works the same as in traditional genetic search, one parent is selected at random from the population, and one parameter is mutated. Each parameter typically has defined maximum and minimum values, and the mutation takes place anywhere within this range.

- **Average** is also a standard operator in traditional genetic search. Two parents

are selected at random from the population, and a child is generated where each parameter is the average of its parents parameters.

- **Double shot** is an extension to the average operator that provides an improvement by converging faster to local minima and also adding an exploratory component. Two parents are selected at random, but instead of one child, three are generated. The first child is the average of the two parents, the second *lower* child is calculated as follows:

$$lower_i = better_i - (average_i - better_i) \tag{5.1}$$

and the third child, *higher* is calculated by:

$$higher_i = worse_i + (worse_i - average_i) \tag{5.2}$$

where the i^{th} parameter of the *lower* and *higher* children are calculated using the i^{th} parameter of the *better* parent and *worse* parent respectively. In essence, the lower and higher children are generated outside of the parents, equally distant from the average and the distance between either parent and the child outside of it is the same as to the average.

- **Probabilistic simplex** randomly generates a child along a line calculated using the simplex method. Different from other operators, this approach can use a variable number of parents. n parents are selected at random and a line is created through the worst parent and the centroid (or average) of the remaining parents. By selecting a random number, *rand*, between two limits, l_1 and l_2, the i^{th} parameter of the child is calculated by:

$$c_i = worst_i + rand * (centroid_i - worst_i) \tag{5.3}$$

Using this equation, a random value of 1.0 would generate a child that is the centroid, 0 would be the worst, and 2.0 would be the reflection of the worst point through the centroid. For the astronomy application, the limits that

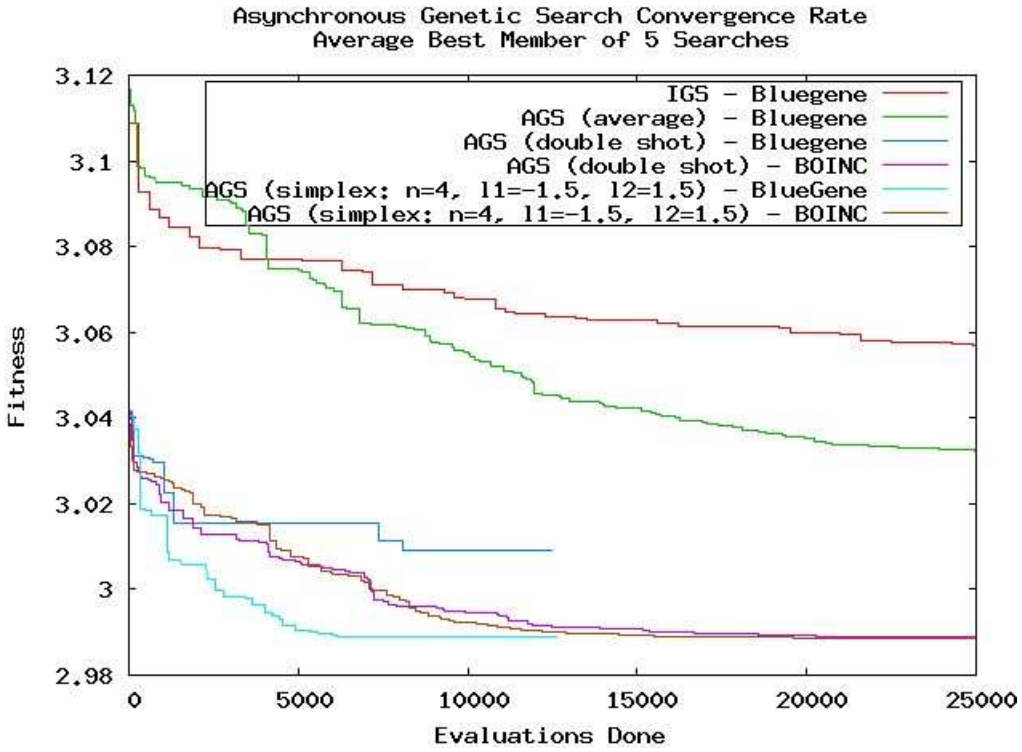

Figure 5.2: A comparison of the different recombination operators for synchronous and asynchronous genetic search.

have been tested are $l_1 = -1.5$, $l_2 = 1.5$ and $l_1 = -0.5$, $l_2 = 2.0$, with the second set providing faster convergence.

Figure 5.2 compares the different recombination operators for genetic search. For all searches mutation was used 30% of the time and the specified recombination operator otherwise. IGS on the BlueGene is standard iterative genetic search and used the average recombination operator. AGS on the BlueGene performs identically to steady state genetic search as every function evaluation was calculated in parallel on the BlueGene. The MilkyWay@Home project was used to calculate the BOINC results, as this volunteer computing project was highly heterogeneous and faulty, it was not possible to run iterative genetic search using it. This figure shows that using an asynchronous strategy improves convergence rates over the traditional iterative strategy, and that using the double shot and simplex operators provide even more benefit.

5.2 Asynchronous Particle Swarm Optimization

Particle swarm optimization is another population based global optimization method, which makes it easily applicable to the asynchronous search framework presented in this chapter. However, as opposed to genetic search, especially the steady state variant, particle swarm is much more iterative in nature. Where in genetic search, individuals are easily created and removed from the population, particle swarm takes a fixed number of particles and updates these iteratively by moving them in the same previous direction and pulling them towards the globally best position found, and each particles' locally best position found. Because of this, for the particle swarm method to be used by the asynchronous search framework, some modifications need to be made.

Asynchronous particle swarm works as follows. The search is initialized by having positions of particles generated at random with zero velocity until there has been a fitness reported for a possible position of each particle. The server keeps track of each particle's current position and current velocity, generating new positions for particles in a round-robin fashion when work is requested. The newly generated particle is generated identically to the original particle swarm optimization algorithm described in Section 2.3, using the current locally best known position for that particle and the current globally best known position. As opposed to the approaches discussed in the related work (see Section 2.3) that only process one position per particle at a time, this approach continues to generate new future positions for particles and send them to workers, updating a particles current position without knowing the fitness of the previously generated points. Using this approach, multiple positions for a single particle can be calculated concurrently, and the search does not need to wait for un-reported fitnesses. When a worker reports the fitness of a particle, it also reports the position and velocity of the particle reported. If the fitness of the reported particle is better than that particle's locally best found position, that position is updated, and the velocity of the particle is reverted to the reported velocity. If the fitness is the globally best found fitness, the position of the global best particle is updated as well.

In this way, asynchronous particle swarm performs nearly identical to tradi-

tional particle swarm when the number of processors used is less than the number of particles and there are no faults, however it can also scale to very large systems by letting workers evaluate possible future positions of a particle. For a large number of workers, the search is more exploratory, examining many possible future positions of a particle assuming the local and global best positions have not been updated. The approach is also resilient to unreported results, as more future positions of a particle are generated until one is found which improves that particle's locally best found position. Additionally, as many generated positions for particles do not improve the global best fitness of the swarm, or a particle's local best fitness, this strategy in a sense lets the search progress faster by generating multiple future positions of particles which can be evaluated concurrently.

5.3 Asynchronous Differential Evolution

Differential evolution has similarities between both genetic search and particle swarm optimization, utilizing multiple parents and recombination (reproduction in genetic search), as well as local and global population information (cognitive and social knowledge in particle swarm optimization). Similar to particle swarm optimization, the current value of an individual is used to generate that individual's next value, however differential evolution uses a monotonically improving strategy on an individual basis.

Differential evolution also has a large suite of different equations to generate the next iteration of an individual (see Section 2.1), which shows varying suitability for different test equations. The *rand/1/bin* and *best/1/bin* variants, for example, seem particularly robust with quick convergence for a wide range of test functions in both traditional single population DE [62] as well as parallel DE with multiple populations [88]. However, both these approaches have iteratively updated populations, so it is an open question whether this will hold true with asynchronously updated individuals – or if other variants will either be less influenced by asynchronous updates or gain benefit from the increased search area that asynchrony can provide.

Both *best/N/bin* and *rand/N/bin* variants of differential evolution were imple-

mented for the asynchronous version. The asynchronous algorithm works similarly to asynchronous particle swarm. New members are generated in a round robin fashion from the population, using the specified recombination strategy (as described in Section 2.1). Unlike particle swarm optimization, individuals are not changed when a new member is generated. Individuals are only updated when a member is reported for that individual with a better fitness value. In this way the individuals evolve with monotonically improving values, as with traditional differential evolution.

CHAPTER 6
Simulating Asynchronous Optimization

While there are different computationally intensive applications available that provide incentive to use this kind of search method and serve as a proof of concept (for example the MilkyWay@Home project [71]) utilizing these to examine the different aspects of asynchronous search is far from exhaustive. However, there are a large number of computationally inexpensive test functions that have challenging search spaces with multiple local minima – for example the Ackley, Griewank, Rastrigrin, and Rosenbrock functions as found in related work [57, 31, 9]. The sphere test function was also used as an easy to solve well formed optimization problem with a single minimum. The optimization test functions used are as follows:

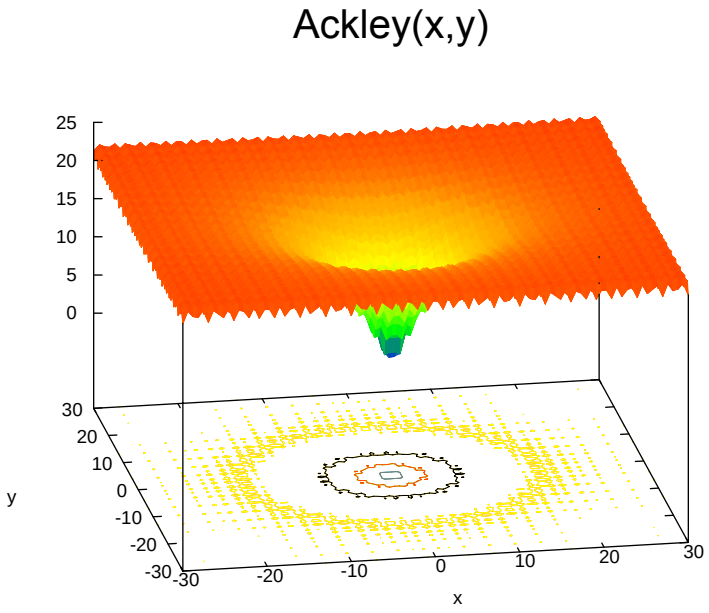

Figure 6.1: The Ackley test function with two input parameters (x, y). This test function has many local minima and a single global minimum at 0,0. Unlike the Griewank and Rosenbrock functions, it converges sharply to the global minimum with a flat external surface.

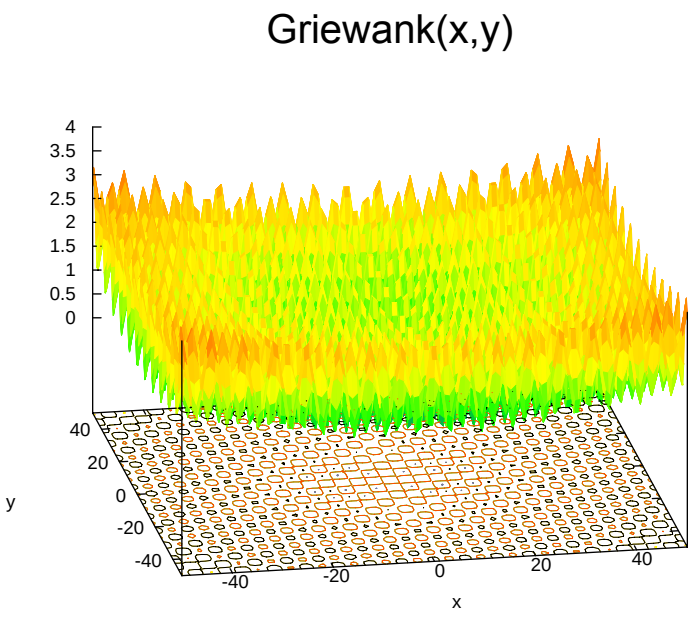

Figure 6.2: The Griewank test function with two input parameters (x, y). This test function has many local minima and a single global minimum at 0,0. This has a shallow curvature with less pronounced peaks and valleys than the Rosenbrock function.

- The **Sphere** function is a simple test function. It has a single minimum with fitness 0 when all input parameters are 0 (see Figure 6.5). The range of parameters used for this optimization problem was from -100 to 100.

$$f_{sphere}(\vec{x}) = \sum_{i=1}^{N} \vec{x}_i^2 \tag{6.1}$$

$$f_{sphere}(0, ..., 0) = 0 \tag{6.2}$$

- The **Ackley** function is a more challenging test function with many local minima. There is a single global minimum with fitness 0 when all input parameters are 0 (see Figure 6.1). It becomes flat near the edges of the problem space and converges sharply to the global minimum. The range of parameters used for this optimization problem was from -32 to 32.

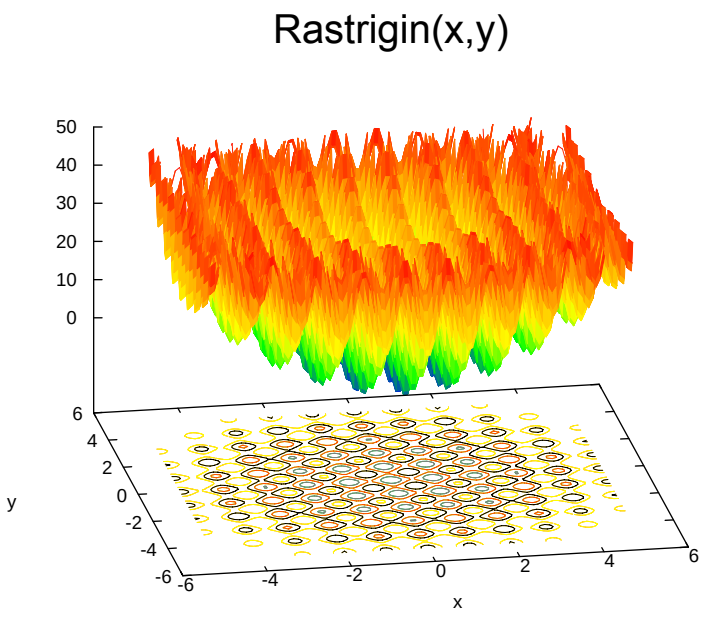

Figure 6.3: The Rastrigin test function with two input parameters (x, y). This test function has many local minima and a single global minimum at 0,0. It has a steeper curvature with more pronounced peaks and valleys than the Griewank function.

$$f_{ackley}(\vec{x}) = 20 + e - 20e^{-0.2\sqrt{\frac{1}{N}\sum_{i=1}^{N}\vec{x}_i^2}} - e^{\frac{1}{N}\sum_{i=1}^{N}cos(2\pi\vec{x}_i)} \tag{6.3}$$

$$f_{ackley}(0, ..., 0) = 0 \tag{6.4}$$

- The **Griewank** function is another challenging test function with many local minima. It has a shallow curvature and a single global minimum with fitness 0 when all input parameters are 0 (see Figure 6.2). The ranges of parameters used for this optimization problem was from -600 to 600.

$$f_{griewank}(\vec{x}) = \frac{1}{4000}\sum_{i=1}^{N}\vec{x}_i^2 - \prod_{i=1}^{N}cos(\frac{\vec{x}_i}{\sqrt{i}}) + 1 \tag{6.5}$$

$$f_{griewank}(0, ..., 0) = 0 \tag{6.6}$$

- The **Rastrigin** function is similar to the Griewank function. However, it has

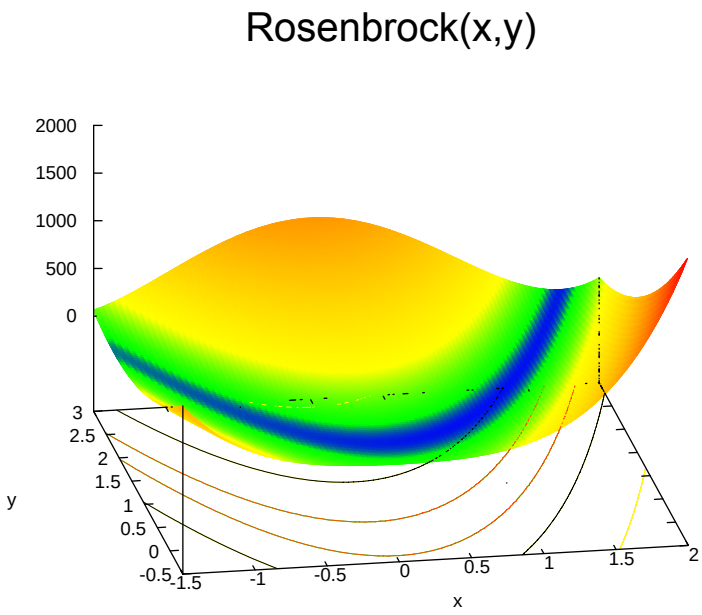

Figure 6.4: The Rosenbrock test function with two input parameters (x, y). This test function has a long flat valley with a single minimum at 1,1.

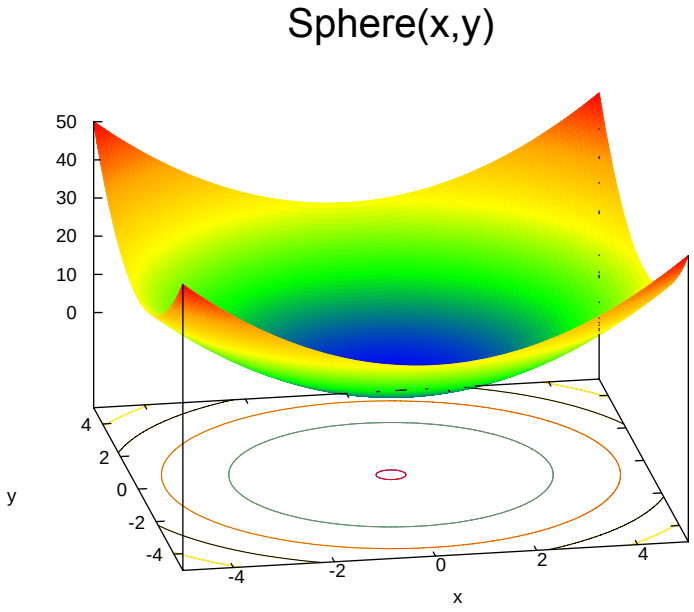

Figure 6.5: The sphere test function with two input parameters (x, y). This is a simple test function with a single minimum at 0,0.

a less shallow curvature and more pronounced peaks and valleys. The global minimum has a fitness of 0 when all input parameters are 0 (see Figure 6.3. The ranges of parameters used for this optimization problem was from -2π to 2π.

$$f_{rastrigin}(\vec{x}) = \sum_{i=1}^{N}[\vec{x}_i^2 - 10cos(2\pi\vec{x}_i) + 10] \tag{6.7}$$

$$f_{rastrigin}(0, ..., 0) = 0 \tag{6.8}$$

- The **Rosenbrock** function has no local minima, however it has a very flat valley with only a single minimum. The minimum has a fitness of 0 when all input parameters are 1 (see Figure 6.4). The ranges of parameters used for this optimization problem was from -30 to 30.

$$f_{rosenbrock}(\vec{x}) = \sum_{i=1}^{N}[\vec{x}_{i+1} - \vec{x}_i^2)^2 + (\vec{x}_i - 1)^2] \tag{6.9}$$

$$f_{rosenbrock}(1, ..., 1) = 0 \tag{6.10}$$

To establish more stringent analysis of asynchronous search strategies and the effect of heterogeneity, the optimization framework presented in Chapter 4 was extended with a simulated computing environment that can be used in place of the different distributed computing environments. This environment can simulate asynchrony and quickly evaluate different asynchronous search methods using various test functions. The simulated computing environment requests parameter sets to be evaluated then returns those results after a simulated amount of time (or not at all). In addition to testing the search methods on functions known to be difficult to find a global optimum for, this also provides a method to evaluate the effect of asynchrony in a controlled environment. This allows the asynchrony to be controlled, for example, the number of updates to the population that occur before the result of a parameter set is reported can be generated through different probability distributions and the minimum and maximum update times for parameter sets can be fixed. It is also possible to have the simulation discard parameter sets, simulating

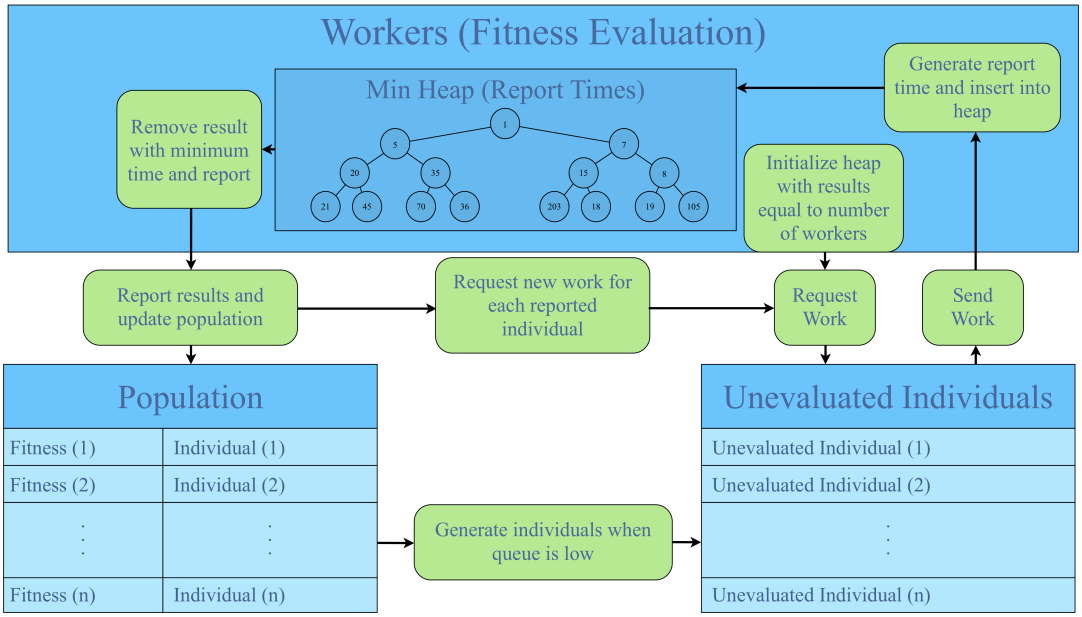

Figure 6.6: The simulated evaluation framework. A simulated environment is used instead of different distributed computing environments. Computed results are stored in a heap and inserted into the population in order of their simulated report time.

failures.

The simulation framework consists of two basic parts (see Figure 6.6). The first allows users to specify templates which control the amount of asynchrony and fault rates in the system. Users can specify the amount of time it takes for results to be calculated and the number of faults that occur. The amount of time it takes for a result to be calculated is probabilistic, specifying the percentage chance for a result to take the time between a minimum and maximum given time and a probabilistic distribution function. Currently, uniform and gamma distributions are implemented. The gamma distribution was chosen because it typically is used to model waiting times, which is ideal for this type of simulation. Multiple distributions can be used, which is important in modeling BOINC-like environments which may have two separate result trip time distributions if different types of hardware are used, such as CPUs and GPUs (see Section 8.1).

The other part is the simulated evaluation environment. The simulations run with a specified number of workers, allowing the size of the simulated computing environment to be modified. A heap is used to handle results that are currently being calculated by the simulated environment. This allows insert and removal of results to be done in $O(log(n))$ time. The heap is initially populated with a number of results equal to the number of workers and each is given a report time that is calculated using the specified distributions in the simulation template. If a result is determined to be faulty it is assigned a random fitness. Following this, the minimum value is removed from the heap and inserted into the search and the current simulation time is set to the report time of that result. A new result is then generated with a new report time, the current time plus a new time generated by the simulation template, and inserted into the heap. This essentially simulates workers reporting a result and requesting a new result.

This simulation framework can represent all the massive scale computing environments as described in Chapter 3. Clusters, supercomputers and even GPUs can be simulated by having a large number of workers and a static result report time. Grids can be simulated by having multiple static report times, each having a probability corresponding to the proportion of processors in each cluster of the grid, and a report time equal to the calculation time on those processors plus the latency to that cluster. Internet computing systems can be simulated by using multiple probability distribution functions representing the different types of hardware used and estimated report times for those volunteers.

CHAPTER 7
Improving Validation Techniques

As volunteer computing systems are open to the public, there always is the risk of malicious users or bad hardware returning false results. This can also be a concern with applications that may occasionally return incorrect results or applications with very high reliability requirements. Typical BOINC applications require every work unit to be verified, and this behavior can be enabled easily within the BOINC architecture. However, validation of every work unit in an asynchronous search setting leads to a large amount of wasted computation.

Asynchronous searches only progress when new individuals are inserted into the population. Individuals with lower fitness are simply discarded. Table 7.1 shows the average number of inserts done over 20 different searches done on Milky-Way@Home using data from Sagittarius stripe 22. The number of inserts for both the first and second 25,000 reported results are shown for asynchronous genetic search, particle swarm optimization and differential evolution (using both best and random parent selection). It becomes apparent from this information that only a small number of results ever make it into the population, less than 4% in the first half of the search and less than 2% in the second half. Additionally, as the searches progress it becomes more difficult to find new good search areas and the number of evaluated individuals inserted into the population decreases.

Additionally, through examining result logs at MilkyWay@Home, less than

Search	0...25,000 Evaluations	25,001 ... 50,000 Evaluations
AGS	701	434
APSO	476	208
ADE/Best	551	221
ADE/Rand	533	161

Table 7.1: The average number of individuals inserted into the population during the given number of evaluations, averaged over 20 searches with different initial parameters.

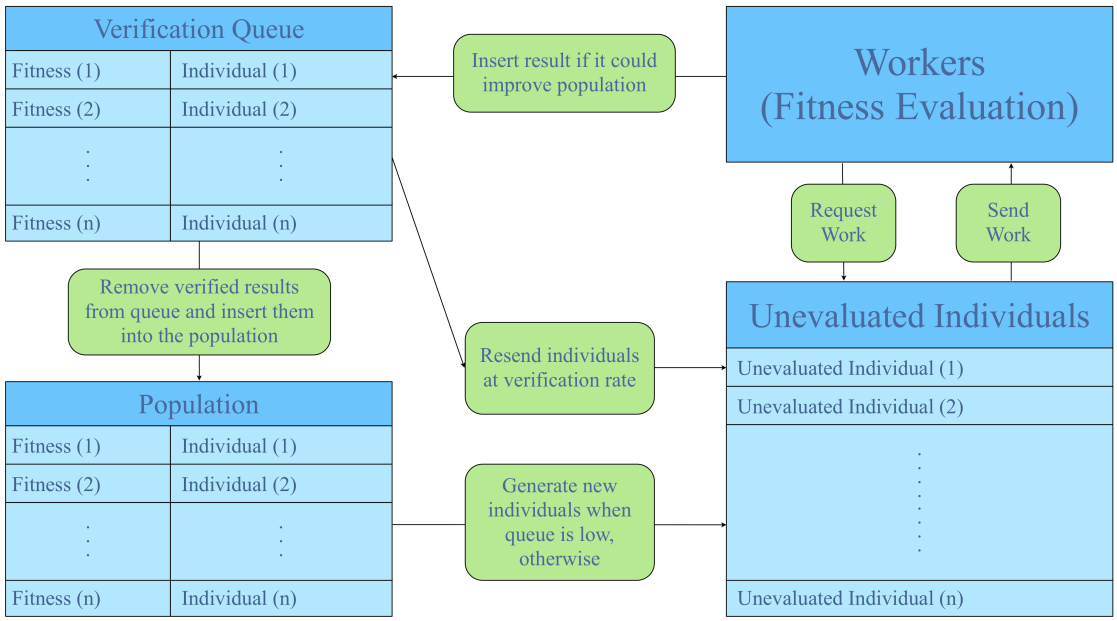

Figure 7.1: The asynchronous optimization strategy updated with validation. Results are stored in a validation queue and only inserted into the population when they are verified. When work is generated, individuals are copied from the verification queue at rate v, and through recombination on the search population otherwise.

1% of verified results are found to be invalid. If every result is validated with a minimum quorum of one, this means that at least 45% of the work units computed are validating results that won't be inserted into the population and would have been discarded anyways. By ignoring the small number of results that when reported would not be inserted into the population, but would be after being validated to something different which potentially could be inserted into the population, we can dramatically increase the amount of useful work being done. However, it is important to note that the search will not progress until better individuals are inserted into its population, so the longer it takes to verify good results, the slower the search will progress. If too many resources are devoted to the evaluation of new results, the number of results waiting validation might grow faster than they are validated or the search may progress extremely slow.

Figure 7.1 describes the changes made to the asynchronous optimization strat-

egy used for validation. Results that could potentially improve the search population are stored in a validation queue. When new work is generated, individuals in the validation queue are copied and sent out at a verification rate, otherwise new individuals are sent as typically generated by the search from the population. Only after an individual in the verification queue has been verified by reaching a quorum is it removed from the verification queue and inserted into the search population. This strategy makes it possible to tune the amount of resources devoted to verification by increasing or decreasing the verification rate.

To implement this, the BOINC assimilator and validator needed to be merged, as in the standard BOINC package they are separate daemons (see Section 3.3.1). In the asynchronous optimization framework described in Chapter 4, search populations are controlled by the assimilator which generates new work units from them. These daemons needed to be merged to allow the validator access to the population data in order to determine which individuals need to be verified. The merged assimilator and validator not only keep information about various searches populations, but also their queues of work units awaiting verification. A tunable verification rate, v, determines the percentage of work units that are generated to verify results waiting in the queue.

The verification queue can be optimized differently for asynchronous genetic search than for asynchronous particle swarm optimization and differential evolution. In order to prevent the verification queue from growing too rapidly, especially in the early stages of the search where almost every returned result needs to be verified due to an initially empty population, the verification queue can be trimmed whenever a new result is verified. In the case of genetic search, when a newly verified result is inserted into the population, any results waiting to be verified with fitness lower than the worst value in the population (assuming the population is full) can be removed, as they will not be inserted into the population even if they are verified. For asynchronous particle swarm and differential evolution, a verification queue can be kept for each individual, so when a result is verified, all other results with lower fitness waiting in an individual's queue can be removed as they are no longer needed.

CHAPTER 8
Results

Asynchronous optimization was examined using both simulated and real massive scale computing environments. Section 8.1 details the different simulated environments used and how asynchronous optimization was affected by heterogeneity and scalability in these environments. Section 8.2 presents results for using asynchronous optimization on the MilkyWay@Home volunteer computing project as well as the effect of the validation strategy presented in Chapter 7.

8.1 Simulation Results

Three different types of environments were simulated to evaluate asynchronous optimization: a homogeneous environment representative of GPUs or supercomputers in Section 8.1.2, a heterogeneous environment constructed to measure the effect of widely varying latencies in Section 8.1.3, and a simulated volunteer computing environment modeled after the MilkyWay@Home volunteer computing project in Section 8.1.4. The different parameters to the optimization methods and the parameters used by the test functions are described in Section 8.1.1.

8.1.1 Optimization and Test Function Parameters

While the population size varied for synchronous optimization, for the asynchronous optimization methods examined in this chapter, identical search parameters were used unless otherwise noted. A population size of 100 was used for all the different search types. For genetic search, a mutation rate of 0.3 was used, and the simplex recombination method was used, with $l_1 = -1.5$, $l_2 = 1.5$ and 2 parents. Particle swarm used an inertia weight, $\omega = 0.5$ (as described in Section 2.3 and Equation 2.14) and $c_1 = c_2 = 2.0$. Differential evolution used binomial recombination with either best or random parent selection, one pair, a recombination scale of 0.5 and a recombination rate of 0.5.

For the different test functions, the Ackley and Sphere problems were opti-

mized using 10 search parameters, while the Griewank, Rastrigin and Rosenbrock problems used 5 search parameters. These test functions are described in detail in Chapter 6.

8.1.2 Simulating Homogeneous Environments

Simulation of a homogeneous environment was done by using a fixed report time for all results. This imitates executing optimization strategies on homogeneous environments like clusters, supercomputers and even GPUs where all function evaluations will complete at the same time. Using this environment, the heap used by the simulation essentially acts in a first-in-first-out manner for an asynchronous search or will complete a population of a synchronous search at the same time, assuming the population size is equal to the number of simulated workers.

Using this simulated environment it was possible to compare the scalability of synchronous and asynchronous searches. To simulate a synchronous search on a homogeneous environment, the population size was set to the number of simulated workers, while for asynchronous search the population size was fixed at 100. Figures 8.1, 8.2, 8.3, 8.4, and 8.5 show average the number of iterations taken for 10 searches to solve the Ackley, Griewank, Rastrigin, Rosenbrock and Sphere functions (described in Chapter 6) for both synchronous and asynchronous search. A solution was said to be found when the best fitness in the search was within 10^{-10} of the minimum function value, which is 0 in the case of all the test optimization functions. An iteration was the amount of time it took for every simulated worker to report their result (which is identical on a homogeneous environment). In this way the number of iterations to solution was equal to the number of times results were sent out to all processing units and then reported back to the master in a homogeneous environment. Using iterations gives both an idea of the time to solution and the number of evaluations taken to reach the solution, as the time to solution is the number of iterations multiplied by the function evaluation time plus latency, and the number of evaluations to solution is the number of workers times the number of iterations. For all these figures, the number of simulated workers was increased from 100 to $100,000$.

For these figures, a lack of data points means that the particular search could not find a solution, except for genetic search with a population size $100,000$ which simply performed too slow to find a solution as inserts in genetic search take $O(log(n))$ time, as opposed to $O(1)$ time for particle swarm and differential evolution. When a search could not find a solution to a problem, this was due to the population being stuck in local minima and not being able to escape them to find the global minima.

It is interesting to note that for the Ackley, Rastrigin and Rosenbrock functions, neither asynchronous nor synchronous differential evolution using the best member as a parent (ADE/best) could find the solution with a population size of 100, however once the population size was increased to $1,000$ for synchronous search it was possible to find a solution. For these optimization problems, a population size of 1000 was used for ADE/best.

In general for all the test functions, asynchronous search scaled significantly better than synchronous search, which either took a similar number of iterations to reach a solution as the number of workers increased, or actually took longer to reach a solution with more workers. Of all the synchronous searches, particle swarm optimization seemed the best able to scale to larger environments, however except for PSO in the Rosenbrock test function, asynchronous search took less iterations to reach a solution as the size of the environment increased.

8.1.3 Simulating Heterogeneous Environments

Simple heterogeneous environments were simulated to see the effect an increased variety of result report times had on the number of evaluations and time to convergence. A fixed computation time of 1 was assigned for each result, and a reporting time simulating heterogeneous latency was added to this. For each search, $10,000$ simulated workers were used. The report time range was increased from 1 to $1,000$ and for each result the report time was generated uniformly between 0 and this range, given the fixed computation time c and the maximum report time range l_{range}, and a uniformly distributed random variable between 0 and 1, r, the report time is calculated:

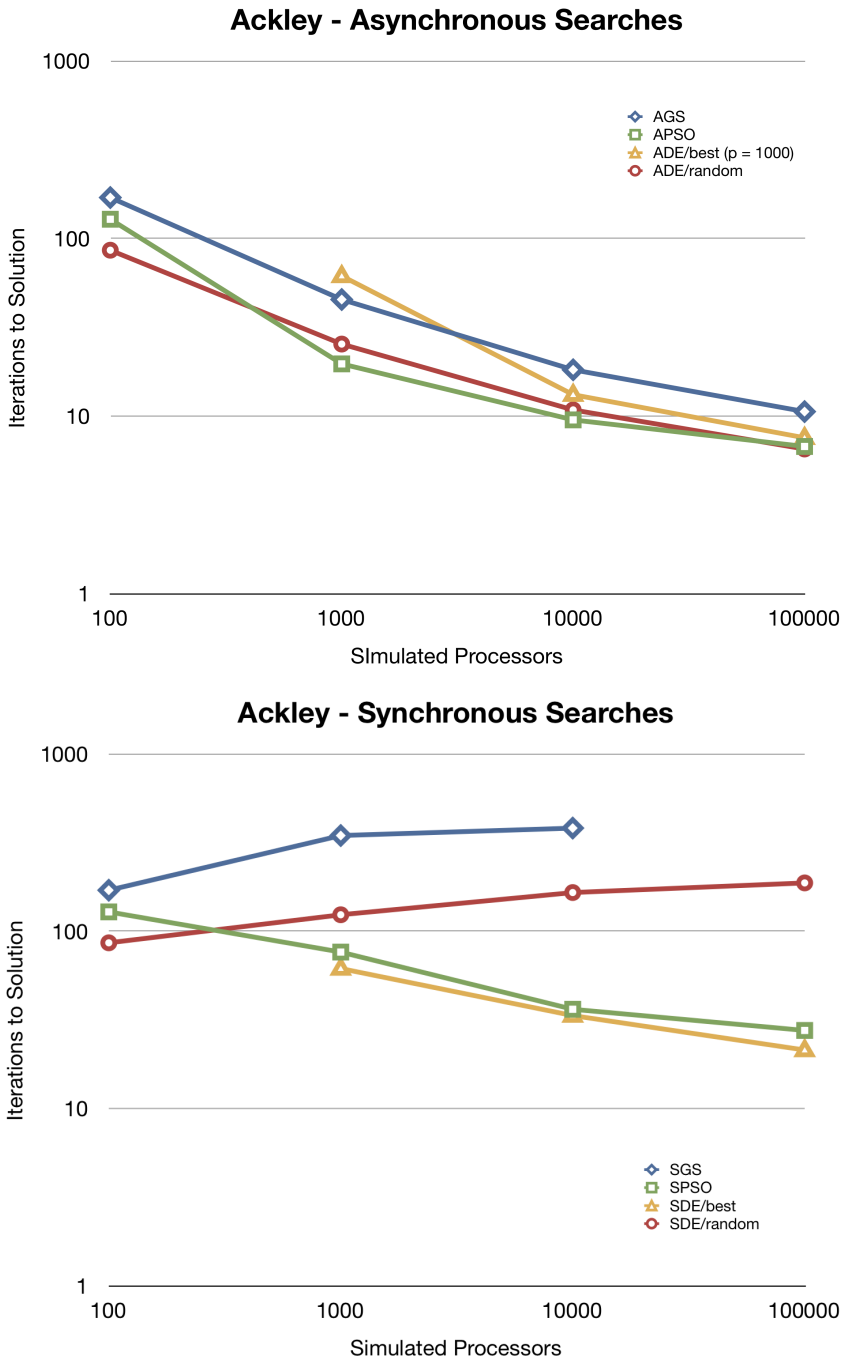

Figure 8.1: Number of iterations to solution for different synchronous and asynchronous optimization strategies for the Ackley test function. Asynchronous searches used a fixed population size of 100, except for ADE/best which required a population size of 1000. Synchronous searches used a population size equal to the number of processors.

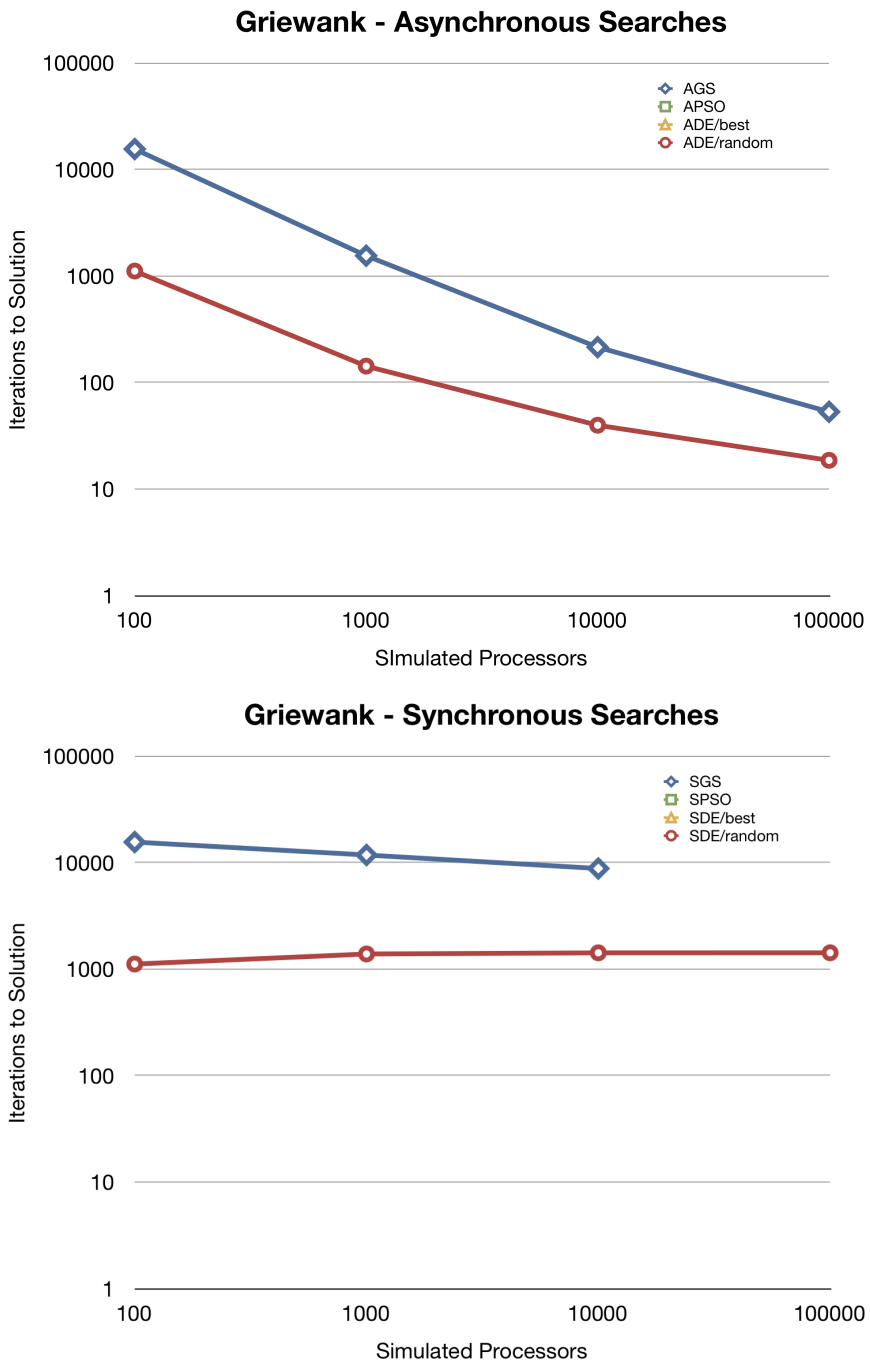

Figure 8.2: Number of iterations to solution for different synchronous and asynchronous optimization strategies for the Griewank test function. Asynchronous searches used a fixed population size of 100, while synchronous searches used a population size equal to the number of processors.

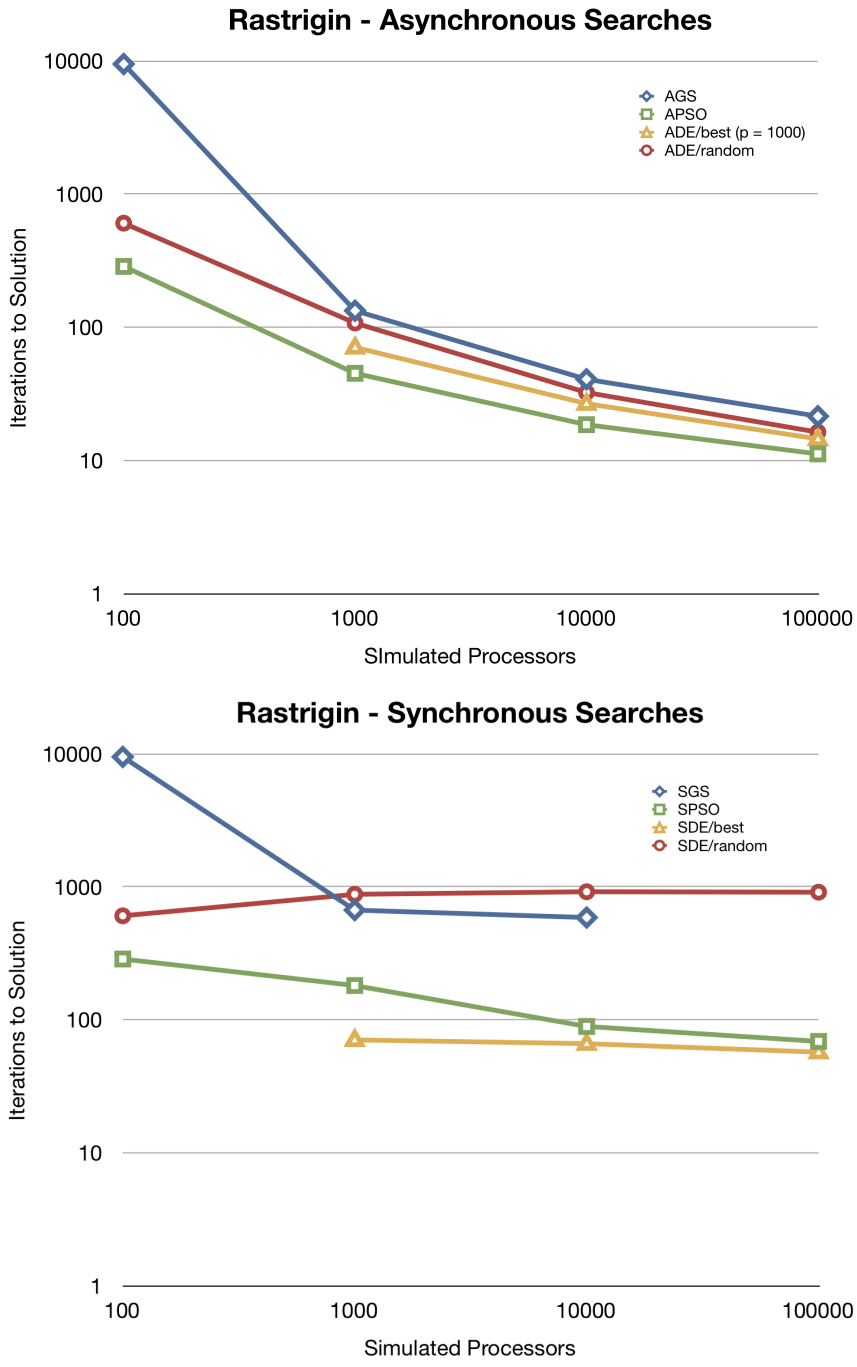

Figure 8.3: Number of iterations to solution for different synchronous and asynchronous optimization strategies for the Rastrigin test function. Asynchronous searches used a fixed population size of 100, except for ADE/best which required a population size of 1000. Synchronous searches used a population size equal to the number of processors.

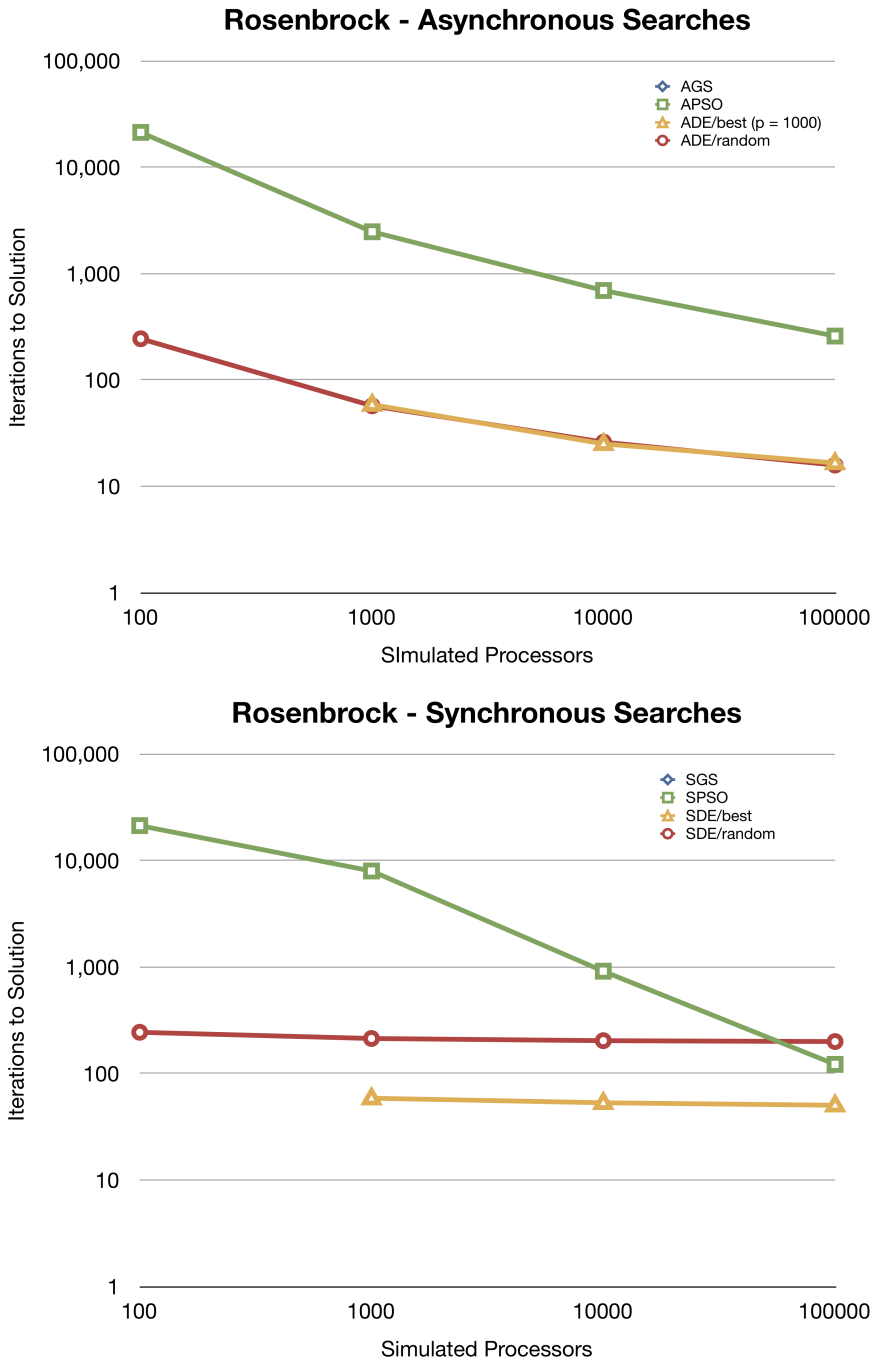

Figure 8.4: Number of iterations to solution for different synchronous and asynchronous optimization strategies for the Rosenbrock test function. Asynchronous searches used a fixed population size of 100, except for ADE/best which required a population size of 1000. Synchronous searches used a population size equal to the number of processors.

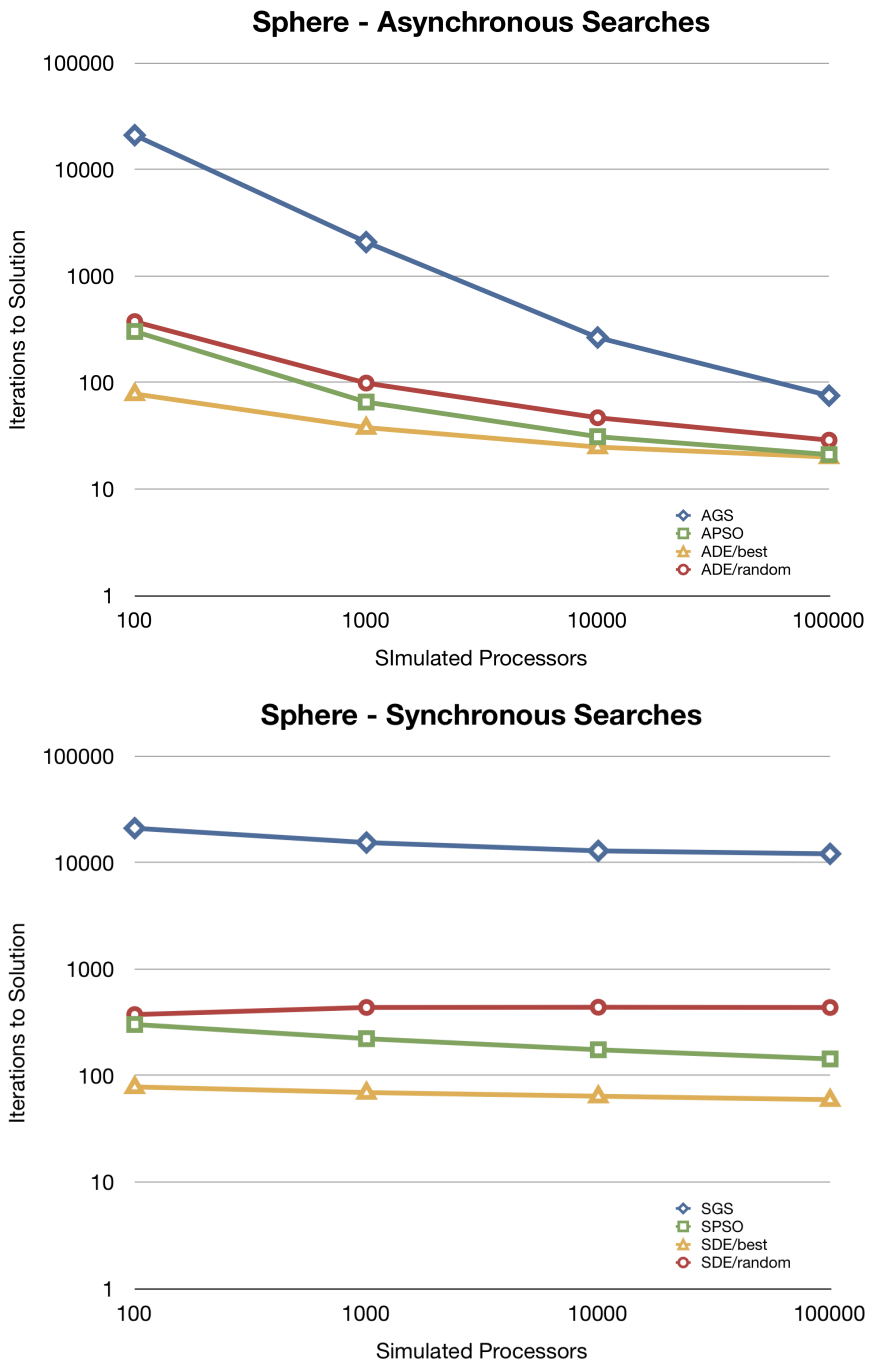

Figure 8.5: Number of iterations to solution for different synchronous and asynchronous optimization strategies for the Sphere test function. Asynchronous searches used a fixed population size of 100, while synchronous searches used a population size equal to the number of processors.

$$time = c + r * l_{range} \tag{8.1}$$

Figures 8.6, 8.7, 8.8, 8.9 and 8.10 show how the different asynchronous optimization methods performed with the five test functions as the range of the simulated latency increased. The results show the average values from 10 different searches. As expected, the time to solution increased as the report time of the results increased, however interestingly the number of evaluations taken to reach a solution decreased as the heterogeneity increased. For some searches this effect tapered off after reaching a certain result report time range, however in general further heterogeneity did not begin to increase the number of evaluations taken to reach a solution.

8.1.4 Simulating MilkyWay@Home

In order to test the scalability of asynchronous optimization on more realistic environments, the time to report results was examined using data from the MilkyWay@Home project. Figure 8.11 shows the frequency of report times for a sample of $10,000$ results reported to the MilkyWay@Home server. The different results were separated into both GPU and CPU results, as GPUs can perform the calculation orders of magnitude faster than CPUs. The result report time is the time it took from the moment a parameter set was generated to the time the parameter set was reported with a result and attempted to be inserted into the search population. The different points on the chart were found by calculating the frequency of a result being reported in 250 second bins, and dividing this frequency by the total number of data points and the bin size.

It was possible to fit the frequency GPU and CPU result times using the gamma distribution (see Equation 8.2), which is commonly used in probability to model waiting times. The gamma distribution function for a random variable x takes two positive input parameters, a shape parameter α and an inverse scale parameter β:

$$gamma(x; \alpha, \beta) = \frac{\beta^{\alpha}}{\Gamma(\alpha)} x^{\alpha-1} e^{-\beta x} \tag{8.2}$$

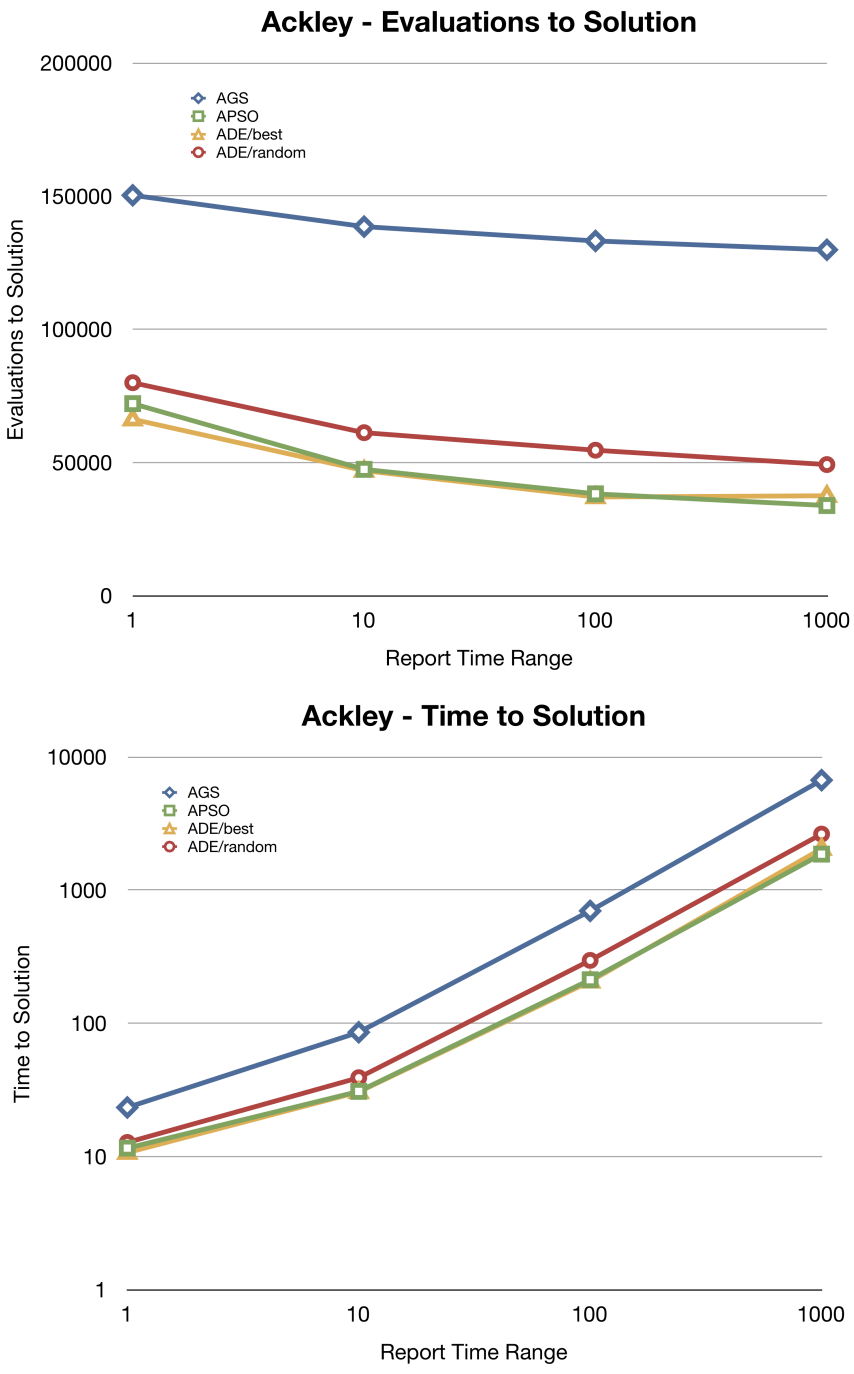

Figure 8.6: Number of evaluations and simulation time to solution for the Ackley test function on a simulated heterogeneous environment. Results were reported using a base latency of 1 plus a uniformly distributed time within the given range.

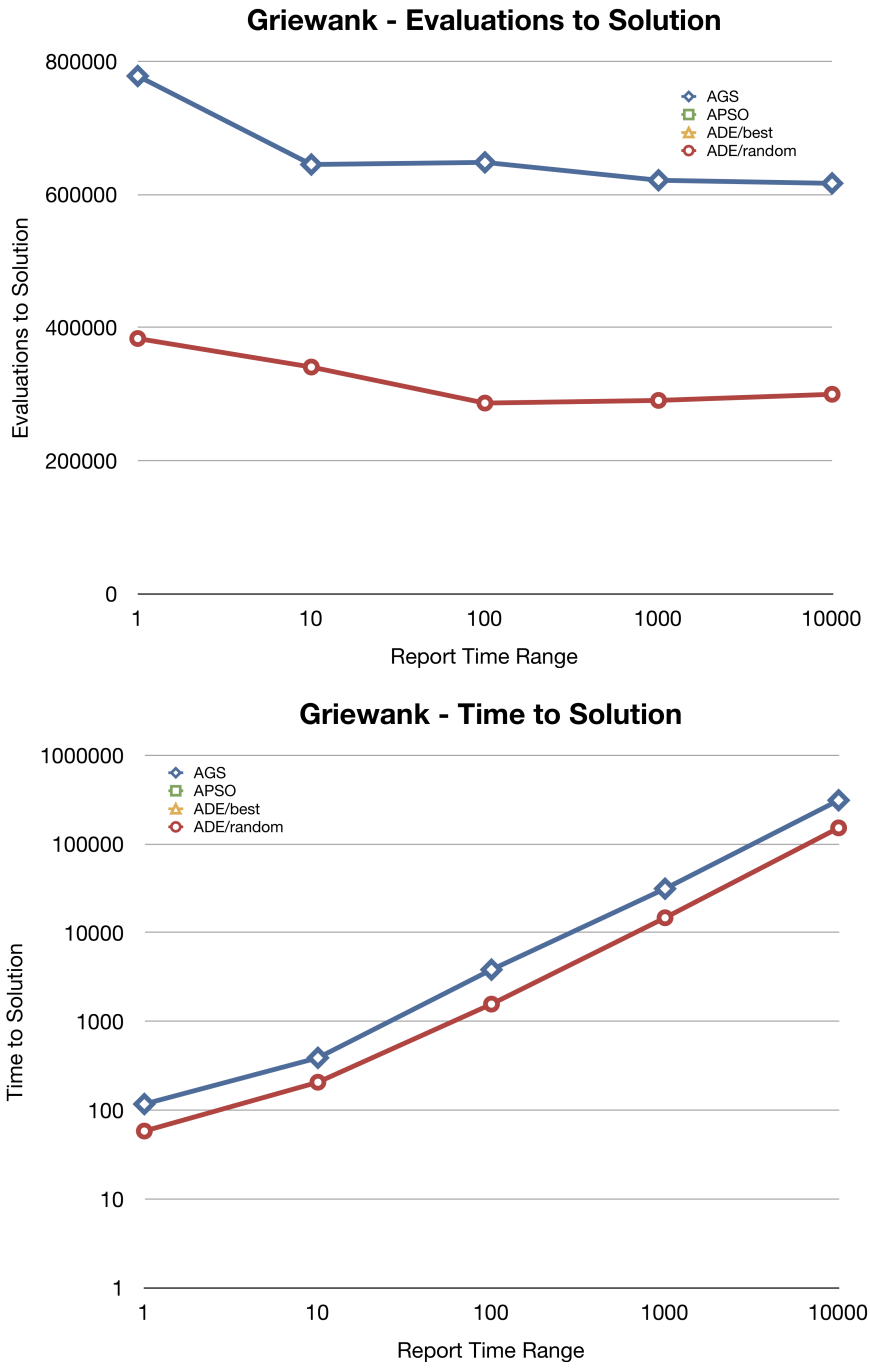

Figure 8.7: Number of evaluations and simulation time to solution for the Griewank test function on a simulated heterogeneous environment. Results were reported using a base latency of 1 plus a uniformly distributed time within the given range.

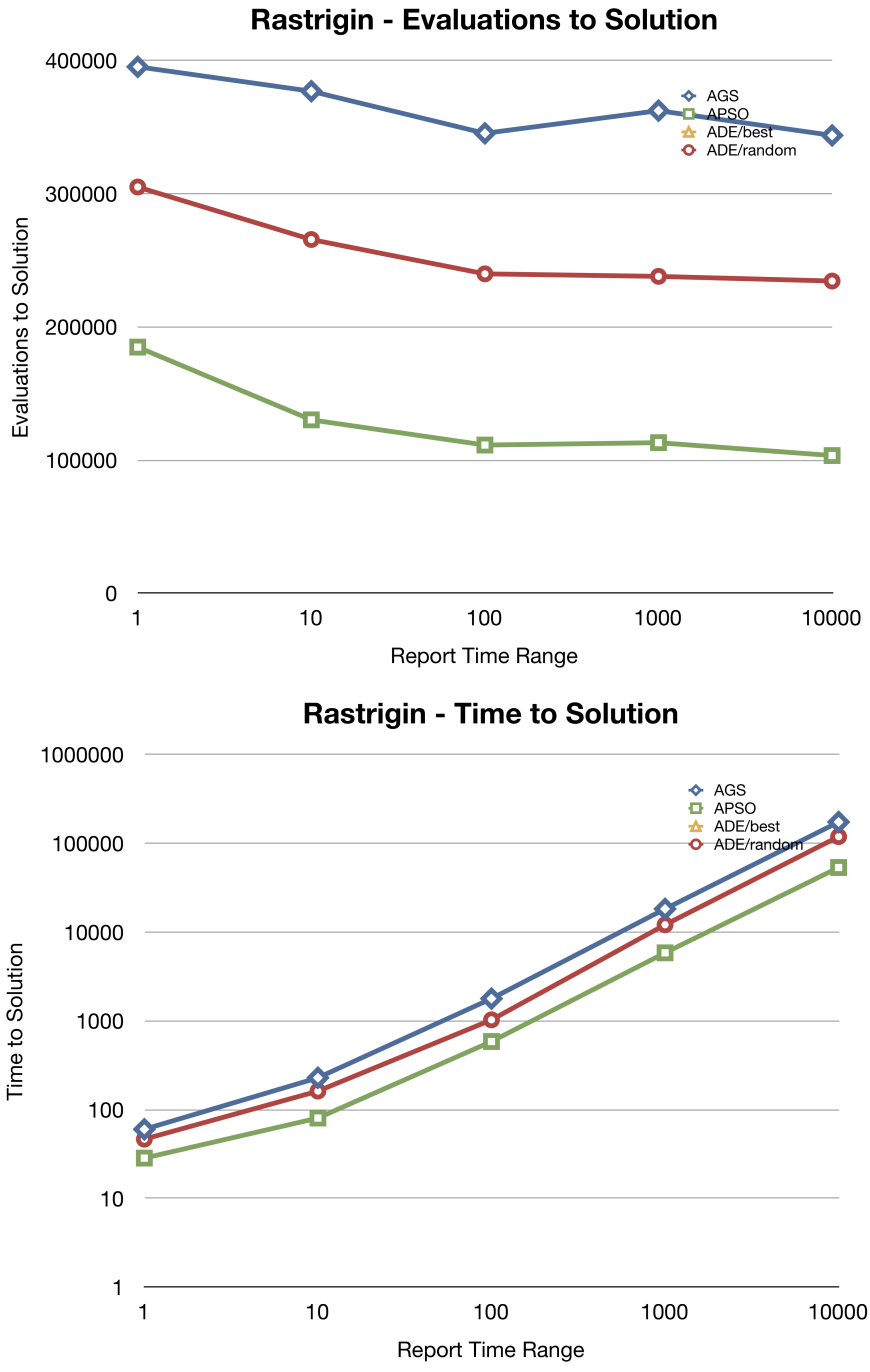

Figure 8.8: Number of evaluations and simulation time to solution for the Rastrigin test function on a simulated heterogeneous environment. Results were reported using a base latency of 1 plus a uniformly distributed time within the given range.

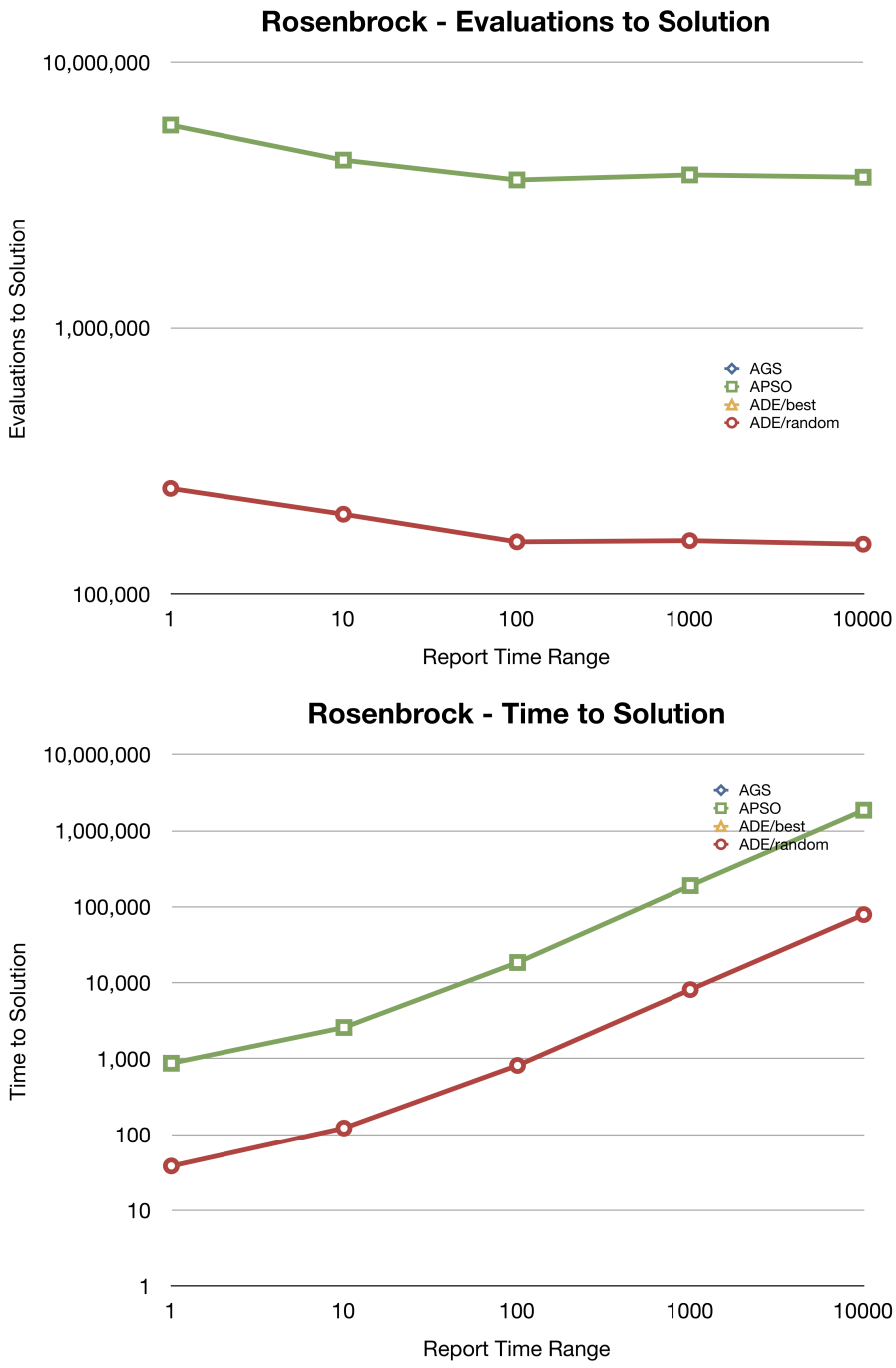

Figure 8.9: Number of evaluations and simulation time to solution for the Rosenbrock test function on a simulated heterogeneous environment. Results were reported using a base latency of 1 plus a uniformly distributed time within the given range.

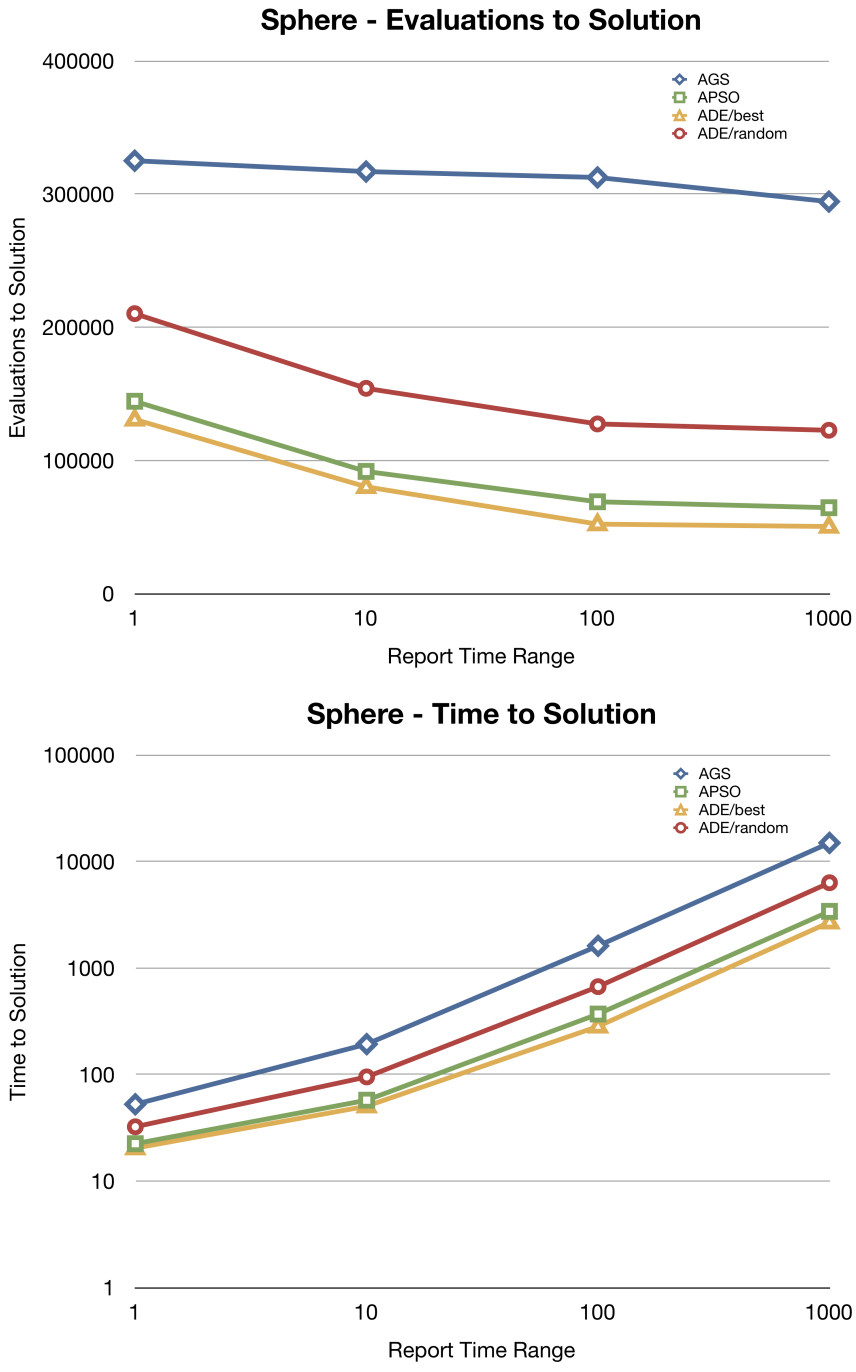

Figure 8.10: Number of evaluations and simulation time to solution for the Sphere test function on a simulated heterogeneous environment. Results were reported using a base latency of 1 plus a uniformly distributed time within the given range.

$$\Gamma(\alpha) = (\alpha - 1)!, \text{ if } \alpha \text{ is a positive integer} \tag{8.3}$$

Figure 8.11 gives the gamma distributions used to model the GPU and CPU wait times. These gamma distributions were used to randomly generate result times with those distributions. As both α and β were integers, random result times could be generated with those distributions as follows:

$$rand_{gamma(\alpha,\beta)} = \frac{1}{\beta}\Sigma_{i=1}^{\alpha}(\ln U_i) \tag{8.4}$$

Where U_i are all uniformly distributed on (0,1] and independent.

As 40% of the results received by MilkyWay@Home were GPU results, the simulation generated result report times using $gamma(3, 1000)$ 40% of the time and with $gamma(2, 12000)$ otherwise. This allowed the simulation to test the scalability of asynchronous optimization on a more realistic heterogeneous volunteer computing environment.

Figures 8.12, 8.13, 8.14, 8.15 and 8.16 show the simulated time to solution and number of evaluations to solution for the five test functions using result times generated with the above gamma distributions. The results show the average results for 10 searches. As expected, in general as the number of simulated workers increased, the number of evaluations to solution increased because more workers were computing results from earlier versions of the evolving population. However, as the number of workers increased, the time to solution decreased, which is a promising result because it means that in a realistic environment, asynchronous search still scales well to very large environments.

Apart from this, a few very interesting things happened in the MilkyWay@Home-like environment. ADE/best, which could not solve the Rastrigin test function with a population size of 100 was able to solve the test function when the number of workers increased to $100,000$. Not only could it solve the problem, but found solutions in less time and evaluations than the other optimization methods tested.

Another unexpected result is that while AGS could solve the Sphere and Griewank problems in both the simulated homogeneous and simple heterogeneous environments, it could not solve the test function in a MilkyWay@Home-like envi-

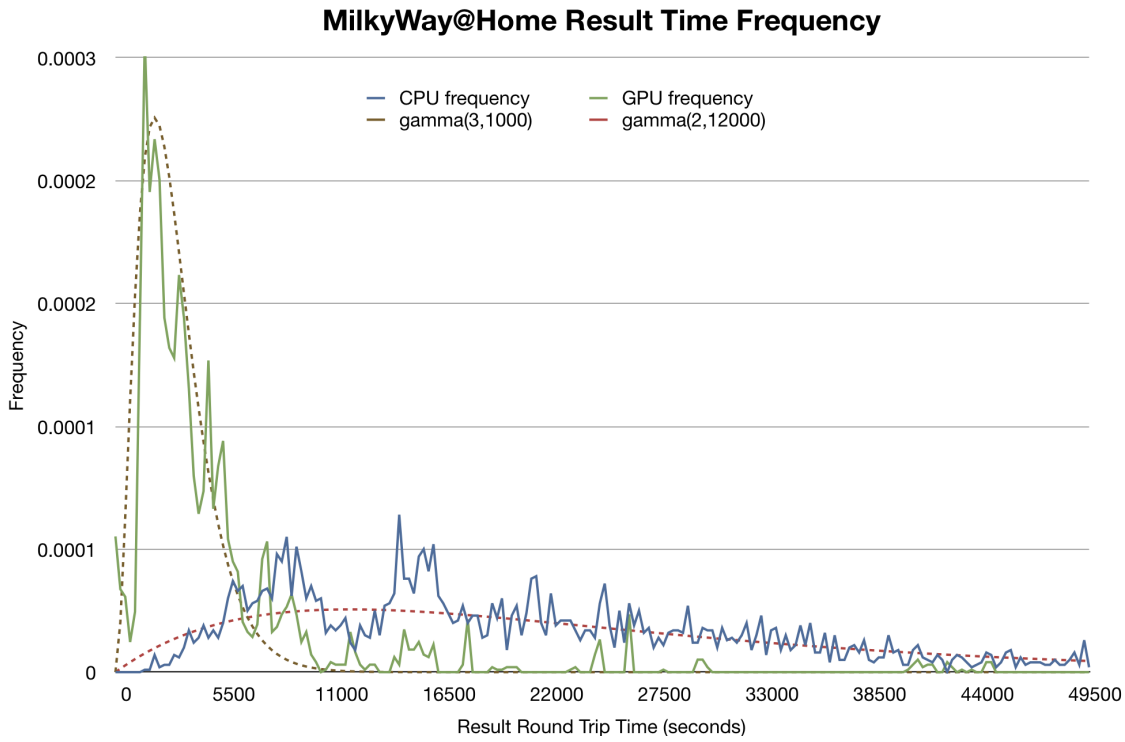

Figure 8.11: Frequency of time taken to download, calculate and report results to the MilkyWay@Home server for GPU and CPU processors.

ronment. This is especially interesting in that the sphere function is well defined and supposedly easy to solve as there is only one minimum. These results seem to indicate that a realistic heterogeneous environment may harm the exploitative ability of asynchronous genetic search.

8.2 Asynchronous Optimization Using MilkyWay@Home

The validation strategy described in Chapter 7 was tested using MilkyWay@Home on Sagittarius Stripe 22 from the Sloan Digital Sky Survey [1]. This problem involves calculating how well a model of three streams of stars and a background function fit a 5 degree wedge of 100,789 observed stars collected along Sagittarian Longitude 55 degrees from Sagittarian Latitude 155 degrees to 230 degrees (for more information about the fitness function readers are referred to [19, 20]). In total there are 20 parameters to be optimized. This model is calculated by a wide variety of hosts.

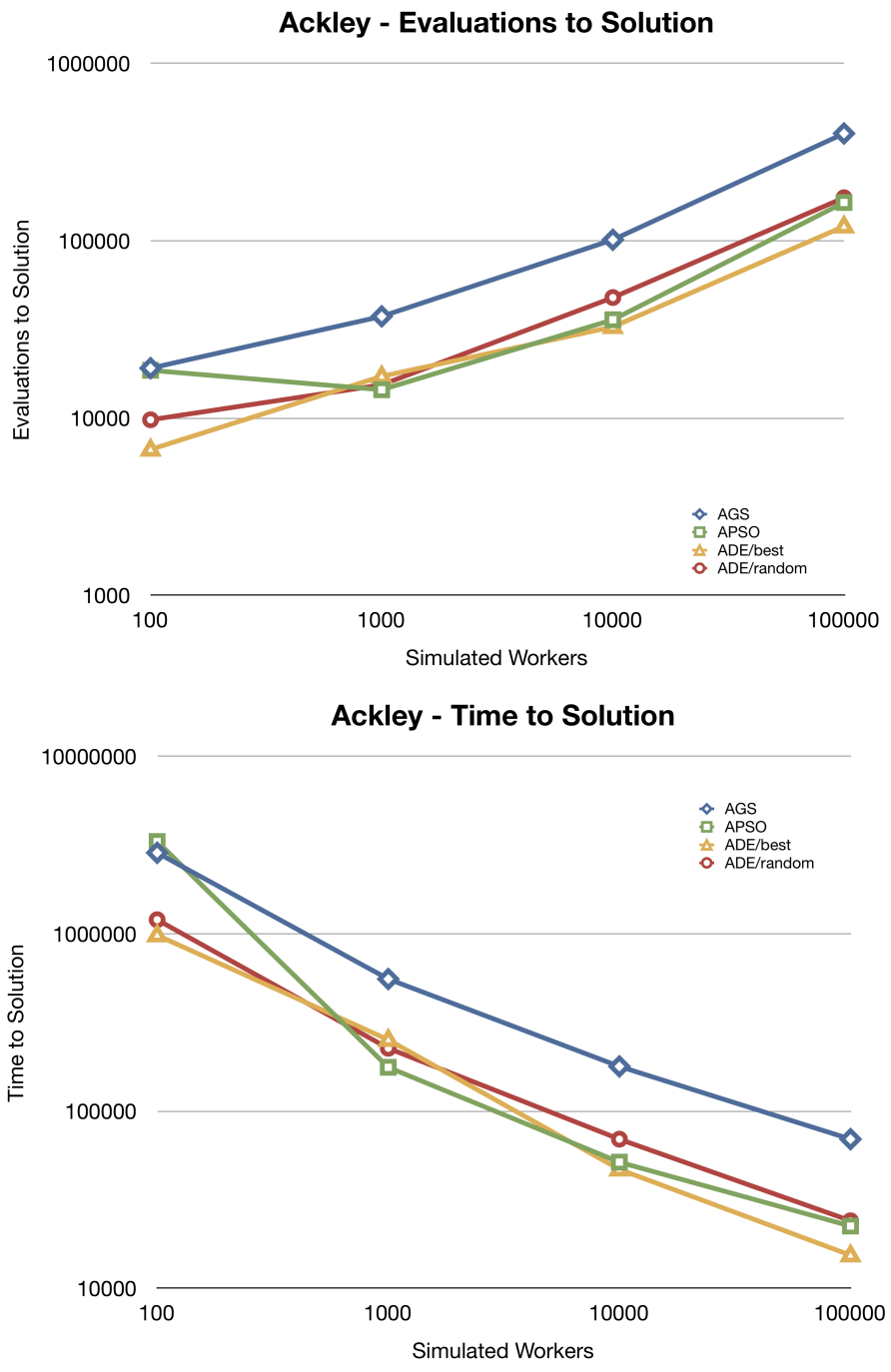

Figure 8.12: Time and number of evaluations to solution for the Ackley test function using a MilkyWay@Home-like simulated environment.

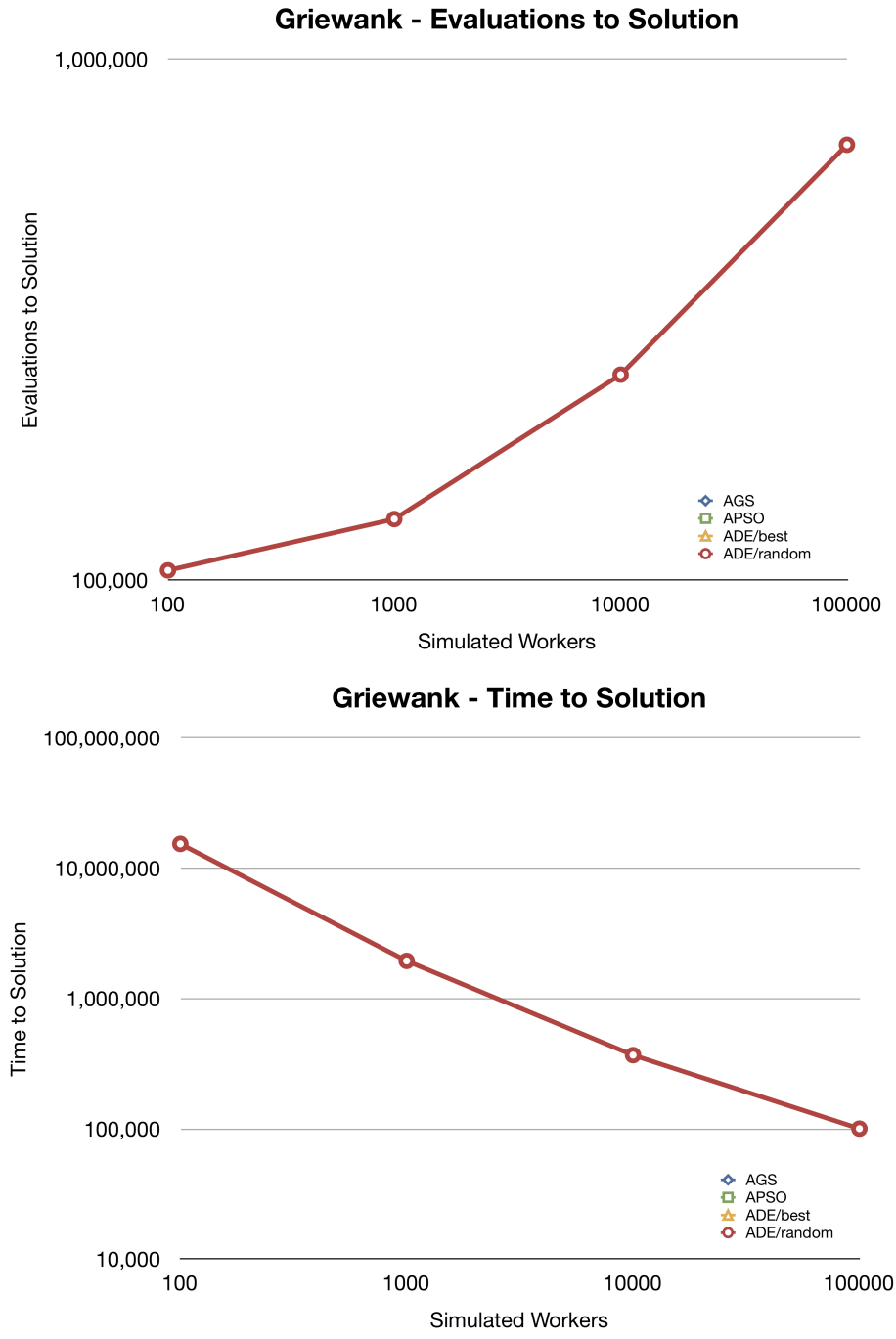

Figure 8.13: Time and number of evaluations to solution for the Griewank test function using a MilkyWay@Home-like simulated environment. It should be noted that AGS could solve this in the other simulated environments.

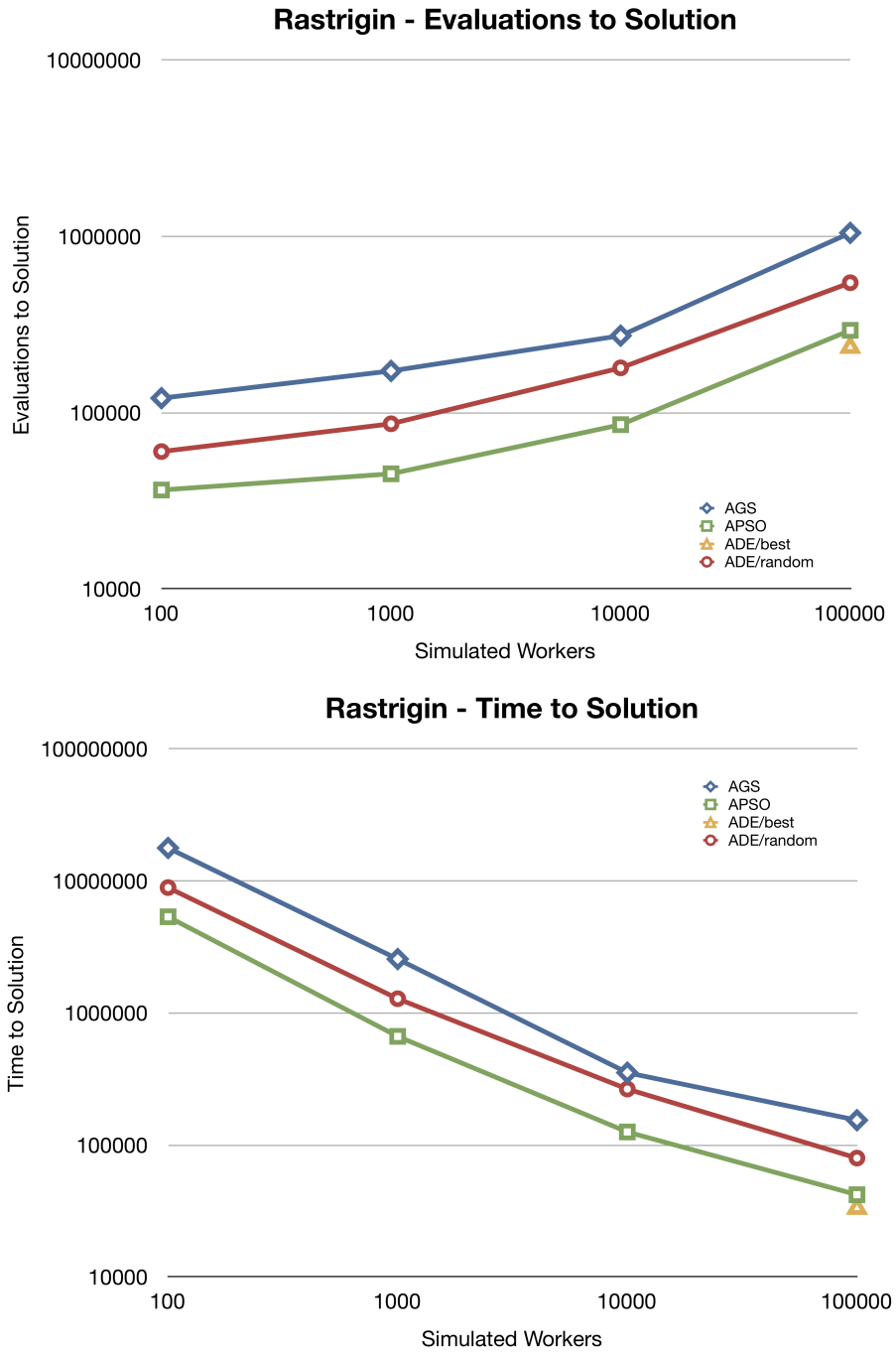

Figure 8.14: Time and number of evaluations to solution for the Rastrigin test function using a MilkyWay@Home-like simulated environment. It should be noted that ADE/best could solve this with $100,000$ simulated workers, but could not in other simulated environments.

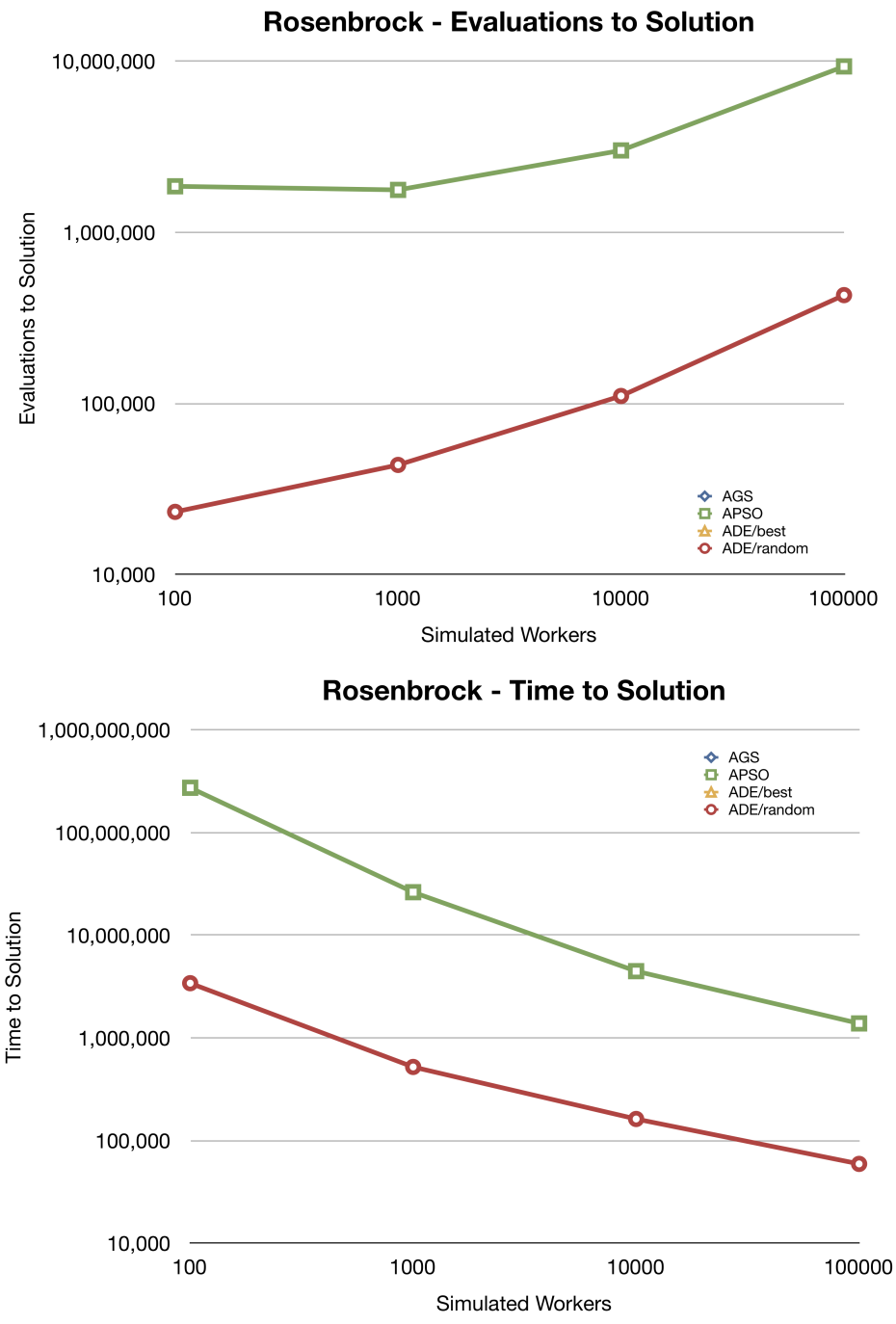

Figure 8.15: Time and number of evaluations to solution for the Rosenbrock test function using a MilkyWay@Home-like simulated environment.

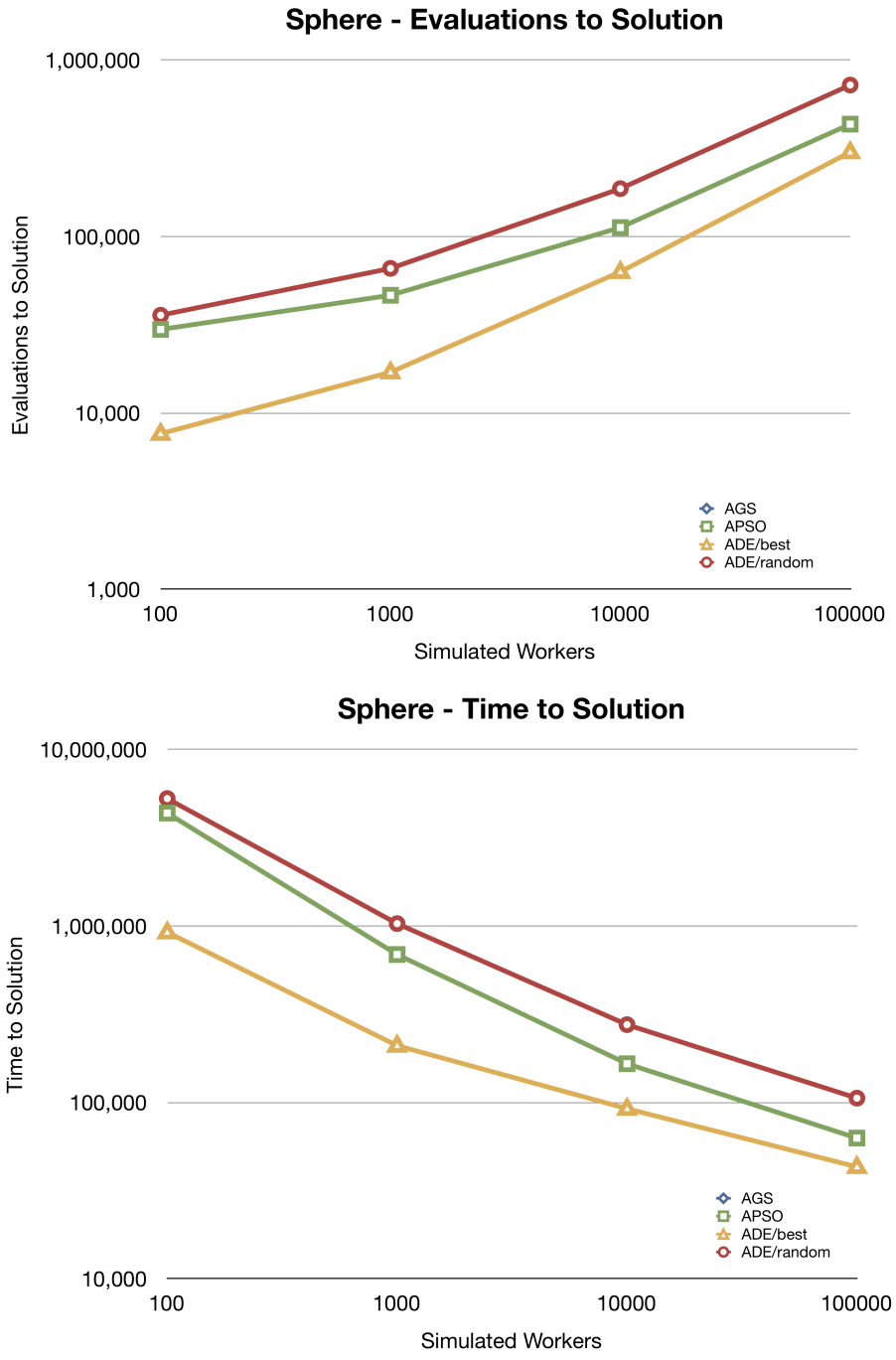

Figure 8.16: Time and number of evaluations to solution for the Sphere test function using a MilkyWay@Home-like simulated environment. It should be noted that AGS could solve this in the other simulated environments.

The fastest high end double precision GPUs can calculate the fitness in under two minutes. High end CPUs require around an hour, and the slowest CPUs can take days. At the time these results were gathered, MilkyWay@Home had approximately 55,000 volunteered hosts participating in the experiments.

8.2.1 Heterogeneity Effects

In order to study the effect of heterogeneity on an uncontrolled environment like MilkyWay@Home, information about the results returned to the server was profiled. Figure 8.17 shows the percentage of results inserted into the population, and the average position these results were inserted into the population as the search progressed for different round trip times. The round trip time was calculated as the number of results that were reported during the time from when the work unit was generated until the time its result was reported and inserted into the population. The population size of the search profiled was 300, so the position inserted ranged from 0 to 299, with 0 being the best individual in the population and 299 being the worst.

The results show that in terms of benefit to the convergence of the search, after the profiling has stabilized, results with all round trip sizes offer similar improvement to the population. With the exclusion of the slowest results (1600 or more results reported during their round trip time), the position the results were inserted into the population had a general trend of improving as the search progressed. It is possible that the slowest results will eventually begin improving if the profiling was not performed for long enough to let those results stabilize. The percentage of results inserted also shows a similar trend of the number of results being inserted into the population improving as the search progresses, with the exclusion of the results with the fastest round trip time. However, these fastest results already have a very high chance to be inserted into the population, and it does not degrade very much. These are very promising results for using asynchronous optimization on the MilkyWay@Home project because they mean that even the slowest hosts still have benefit.

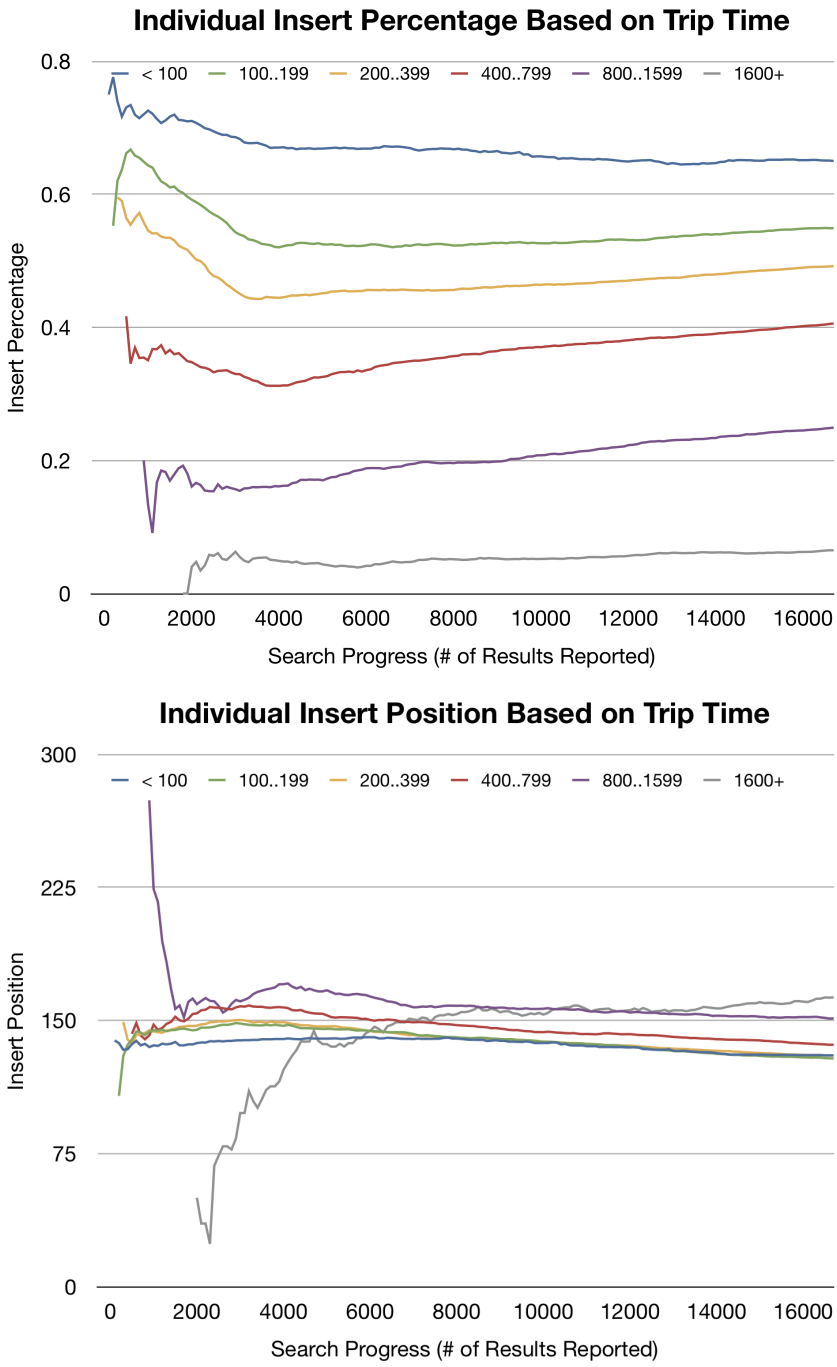

Figure 8.17: Percentage and position of the results inserted into the population for AGS using probabilistic simplex recombination on MilkyWay@Home for results with different round trip times. The round trip time is how many results were reported between the time the work unit was generated and its result was received.

8.2.2 Validation Effects and Search Comparison

Asynchronous differential evolution (ADE), asynchronous genetic search (AGS) and asynchronous particle swarm optimization (APSO) were all tested on Milky-Way@Home using a 50 member population. ADE was tested using both random and best parent selection, and 1, 2, and 3 differential pairs. APSO was tested with inertia weights of 0.2, 0.4, 0.6, 0.8 and 1.0. AGS was tested using 2, 4, 6, 8 and 10 parent individuals as input to the simplex recombination (see Section 5.1 for an in depth presentation of this method). As input to the simplex recombination, $l_1 = -1.5$ and $l_2 = 0.5$ were used as these have been shown to be good values for this problem [87, 29].

Verification rates of 0.3, 0.6 and 0.9 were tested across all these search parameters for pessimistic validation. For each search parameter and verification rate, five searches were performed simultaneously and exclusively by the search manager. This was done to keep the network that the searches were tested on as similar as possible.

Tables 8.1, 8.2, 8.3 and 8.4 shows the best, average and worst fitness across these five searches after 25,000 and 50,000 reported individuals, for all combinations of verification rates and search parameters. Values in boldface represent the best individual found by each search type with the same verification rate after a certain amount of reported individuals, values in italics represent the best values found by a search type over all verification rates. Underlined values show the best fitness found over all three search types.

For APSO, the average best point converged the fastest with $v = 0.3$, while the average worst point converged the fastest with $v = 0.6$. The average average point initially converged faster with $v = 0.6$ for the first half of the search, but then converged faster for the remainder of the search with $v = 0.3$. For ADE (best and random) the average best point converged fastest with $v = 0.6$, and the average average point first converged faster with $v = 0.3$ and then $v = 0.6$. For DE/best, the average worst point converged fastest with $v = 0.3$ and for DE/rand the average worst point converged fastest with $v = 0.6$. For AGS, the average best point first converged fastest with $v = 0.3$ for the first half of the search, but for the rest faster

Asynchronous Particle Swarm				
		25,000 results reported		
		best	average	worst
v = 0.3	w = 0.2	**-3.169272**	*-3.170184*	-3.170872
	w = 0.4	-3.169807	-3.170349	-3.170816
	w = 0.6	-3.169887	-3.170272	*-3.170420*
	w = 0.8	-3.169913	-3.170747	-3.172100
	w = 1.0	-3.171184	-3.172528	-3.174188
v = 0.6	w = 0.2	-3.169545	-3.173018	-3.170252
	w = 0.4	-3.169694	-3.169936	**-3.170110**
	w = 0.6	*-3.169505*	**-3.169815**	-3.170221
	w = 0.8	-3.170305	-3.170817	-3.171458
	w = 1.0	-3.171372	-3.172507	-3.173307
v = 0.9	w = 0.2	-3.170161	-3.171611	-3.173086
	w = 0.4	*-3.169895*	*-3.170542*	*-3.171397*
	w = 0.6	-3.170182	-3.171695	-3.175657
	w = 0.8	-3.171230	-3.172986	-3.177449
	w = 1.0	-3.170701	-3.173401	-3.175725
		50,000 results reported		
		best	average	worst
v = 0.3	w = 0.2	-3.169037	-3.169355	*-3.169679*
	w = 0.4	**-3.167358**	**-3.169100**	-3.170081
	w = 0.6	-3.169128	-3.169579	-3.169930
	w = 0.8	-3.169243	-3.169965	-3.170665
	w = 1.0	-3.170732	-3.171247	-3.171778
v = 0.6	w = 0.2	*-3.168843*	-3.169197	-3.169487
	w = 0.4	-3.169073	-3.169405	-3.169650
	w = 0.6	-3.168870	*-3.169142*	**-3.169374**
	w = 0.8	-3.169647	-3.169900	-3.170217
	w = 1.0	-3.171011	-3.172153	-3.173089
v = 0.9	w = 0.2	-3.169592	-3.170143	-3.171253
	w = 0.4	*-3.169333*	*-3.169583*	*-3.170070*
	w = 0.6	-3.169506	-3.169807	-3.170136
	w = 0.8	-3.170344	-3.171061	-3.171553
	w = 1.0	-3.170329	-3.171741	-3.173602

Table 8.1: This table shows the best, average and worst of the best individuals found by five asynchronous particle swarm searches after 25,000 and 50,000 results reported. Best results for each verification rate (v) and inertia weight (w) are in italics, and best across all verification rates are in boldface. Best across all search methods are underlined.

Asynchronous Differential Evolution (Best)				
		25,000 results reported		
		best	average	worst
v = 0.3	p = 1	*-3.169597*	*-3.170029*	**-3.170256**
	p = 2	-3.169617	-3.170124	-3.170683
	p = 3	-3.169983	-3.170449	-3.171080
v = 0.6	p = 1	**-3.169339**	**-3.169851**	*-3.170350*
	p = 2	-3.170782	-3.171203	-3.171679
	p = 3	-3.170180	-3.170787	-3.171470
v = 0.9	p = 1	*-3.170371*	*-3.171670*	*-3.172434*
	p = 2	-3.171098	-3.172052	-3.172852
	p = 3	-3.170929	-3.173074	-3.176117
		50,000 results reported		
		best	average	worst
v = 0.3	p = 1	*-3.168608*	**-3.168977**	**-3.169291**
	p = 2	-3.169447	-3.169550	-3.169741
	p = 3	-3.169701	-3.169892	-3.170067
v = 0.6	p = 1	**-3.168580**	*-3.169043*	*-3.169805*
	p = 2	-3.169779	-3.169916	-3.170125
	p = 3	-3.170093	-3.170259	-3.170539
v = 0.9	p = 1	*-3.169025*	*-3.170066*	-3.171049
	p = 2	-3.170224	-3.170627	-3.171035
	p = 3	-3.169846	-3.170455	*-3.171027*

Table 8.2: This table shows the best, average and worst of the best individuals found by five asynchronous differential evolution searches after 25,000 and 50,000 results reported. Best results for each verification rate (v) and number of pairs (p) are in italics, and best across all verification rates are in boldface. Best across all search methods are underlined.

with $v = 0.6$. The average worst point converged fastest with $v = 0.3$ and the average average point with first $v = 0.3$ then $v = 0.6$. Over all searches, $v = 0.9$ provided the worst convergence results.

The validation rate, v, had the most dramatic effect on search convergence, compared to all other search parameters. These results show that for this validation strategy, with the exclusion of APSO, generally the best convergence rates were found devoting a rather significant amount of resources ($v = 0.6$) to quickly validate newly found results. Even for APSO, setting $v = 0.6$ resulted in similar convergence

Asynchronous Differential Evolution (Random)				
		25,000 results reported		
		best	average	worst
v = 0.3	p = 1	*-3.170270*	*-3.170390*	**-3.170761**
	p = 2	-3.170443	-3.170859	-3.171600
	p = 3	-3.170553	-3.171094	-3.172143
v = 0.6	p = 1	**-3.169534**	**-3.170287**	*-3.170973*
	p = 2	-3.170319	-3.171231	-3.172264
	p = 3	-3.170887	-3.172007	-3.173013
v = 0.9	p = 1	*-3.172570*	*-3.173701*	*-3.174941*
	p = 2	-3.172753	-3.175244	-3.179046
	p = 3	-3.173753	-3.175168	-3.177439
		50,000 results reported		
		best	average	worst
v = 0.3	p = 1	-3.169623	*-3.169835*	*-3.169986*
	p = 2	*-3.169560*	*-3.169835*	-3.170295
	p = 3	-3.169967	-3.170279	-3.170547
v = 0.6	p = 1	**-3.169510**	**-3.169770**	**-3.169956**
	p = 2	-3.169852	-3.170183	-3.170319
	p = 3	-3.170253	-3.170701	-3.171349
v = 0.9	p = 1	-3.171026	*-3.171628*	*-3.172018*
	p = 2	-3.170894	-3.171979	-3.173157
	p = 3	*-3.170615*	-3.171978	-3.172903

Table 8.3: This table shows the best, average and worst of the best individuals found by five asynchronous differential evolution searches after 25,000 and 50,000 results reported. Best results for each verification rate (v) and number of pairs (p) are in italics, and best across all verification rates are in boldface. Best across all search methods are underlined.

rates to $v = 0.3$. This tends to show that updating the population as fast as possible has a very strong effect on the convergence of the search.

Asynchronous Genetic Search				
		25,000 results reported		
		best	average	worst
v = 0.3	p = 2	-3.171410	-3.173042	-3.174443
	p = 4	-3.170200	-3.171203	**-3.171817**
	p = 6	-3.171318	-3.171887	-3.172633
	p = 8	-3.170104	-3.171009	-3.172346
	p = 10	**-3.169374**	**-3.170770**	-3.172604
v = 0.6	p = 2	-3.171732	-3.173018	-3.174762
	p = 4	-3.170204	-3.171466	*-3.173090*
	p = 6	-3.170994	-3.173666	-3.175648
	p = 8	-3.170366	-3.171647	-3.172820
	p = 10	*-3.169655*	*-3.171382*	-3.173527
v = 0.9	p = 2	*-3.171128*	*-3.175371*	-3.181739
	p = 4	-3.174878	-3.177981	-3.183386
	p = 6	-3.172288	-3.178447	-3.185784
	p = 8	-3.173696	-3.176495	*-3.179121*
	p = 10	-3.182238	-3.182855	-3.183472
		50,000 results reported		
		best	average	worst
v = 0.3	p = 2	-3.170022	-3.170320	-3.170653
	p = 4	-3.169413	*-3.169697*	**-3.170042**
	p = 6	-3.170146	-3.170519	-3.171506
	p = 8	-3.169340	-3.169903	-3.170603
	p = 10	*-3.169057*	-3.169980	-3.171557
v = 0.6	p = 2	-3.170036	-3.170656	-3.171376
	p = 4	**-3.168789**	**-3.169626**	*-3.170054*
	p = 6	-3.169242	-3.171014	-3.172707
	p = 8	-3.169572	-3.170106	-3.170700
	p = 10	-3.169431	-3.170073	-3.170988
v = 0.9	p = 2	*-3.169240*	*-3.172969*	*-3.177777*
	p = 4	-3.171966	-3.174694	-3.180511
	p = 6	-3.170294	-3.176296	-3.184854
	p = 8	-3.172342	-3.175242	-3.178753
	p = 10	-3.180842	-3.181347	-3.181852

Table 8.4: This table shows the best, average and worst of the best individuals found by five asynchronous genetic searches after 25,000 and 50,000 results reported. Best results for each verification rate (v) and number of parents (p) are in italics, and best across all verification rates are in boldface. Best across all search methods are underlined.

CHAPTER 9
Future Work

The work in this thesis is far from exhaustive and also provides a strong foundation for further research in asynchronous optimization. The following sections describe possible avenues of future work to improve the effectiveness and usability of asynchronous optimization.

9.1 Metaheuristics and Hybrid Methods

As shown by the related work, using hybrid methods is a popular strategy for enhancing global optimization methods. Hybridization of genetic search and the simplex method has already been shown to be effective for the astroinformatics application in an asynchronous framework. With knowledge of the different attributes and strengths of the different asynchronous searching methods presented in this chapter, hybridizing them could provide further benefit both in terms of traversing the search space in different ways, which would enhance the searches' ability to find the global optimum, and also by enhancing the searches' convergence to the optima with improved local searching capabilities.

Unfortunately, the complexity of having multiple searches, each with multiple tuning parameters, such as the number of parents used to generate new parameter sets, line search limits, and mutation rate for genetic search, as well as social (global), cognitive (local) weights, and inertia weights for particle swarm optimization, can become prohibitive as manually tuning so many parameters becomes extremely time consuming as typically these are fixed for an entire search. Even optimizing the search parameters for a single search method is time consuming, and can dissuade scientists from using an optimization method for their research. Additionally, it has also been shown that the best values for these parameters may change dynamically over the course of the search [29, 28, 87], so a fixed value still may not provide the best convergence.

With a large pool of operators to choose from, each with different parameters

to tune, dynamically adapting the search as it progresses - metaheuristics - become of particular interest. The FGDO framework is already able to profile the benefit of parameter sets based on how they were generated and how long it took for them to be reported [29]. By utilizing this information, a metaheuristic should be able to tune the parameters of the different operators and select which operators to use. This becomes possible in part because of the computational complexity of function evaluations for these environments allowing the search to take more time in generating new parameter sets.

Previous analysis of the benefit of parameter sets was based on where they were inserted into the population, which determines how much that parameter set improved the population, and chance a generated parameter set had to be inserted into the population depending on how it was generated. These metrics only provide information corresponding to the convergence of the search, not necessarily how well the search is exploring new areas. To help determine the exploratory quality of the proposed operators, profiling the distance of the inserted parameter sets from the population would also be beneficial.

9.2 Simulation and Scheduling

The simulation framework implemented in this work can be used for more than examining the effects of different computing environments by controlling the heterogeneity, size and latency of the simulated workers involved. Of particular interest is studying different scheduling strategies for asynchronous optimization which may improve the time to solution and optimize utilization of resources by assigning certain types of individuals to certain types of workers. For example, it has been shown that certain individuals generated by asynchronous optimization can be more exploratory, looking for new potentially valid areas in the search space, while others are more exploitative, trying to find the minima within an area of the search space. Examining different scheduling strategies in simulated environments may show that there is benefit to sending exploratory individuals to slower less reliable hosts and exploitative individuals to faster more reliable hosts. It also may make sense to initially send exploratory individuals to faster hosts and change this

over time as the search needs to explore less and exploit more.

Scheduling can also play an important role in improving verification strategies. Using the verification strategy presented in this thesis, asynchronous optimization only progresses as fast as results are verified, because the population new individuals are generated from consists only of verified individuals. This strategy could be improved by only sending results that need to be verified to fast reliable hosts, increasing the turnaround rate for validation and reducing the number of duplicate individuals that need to be calculated.

9.3 Validation

The validation strategy presented in this thesis is pessimistic, assuming every individual that could potentially improve the search population is invalid and only using them to update the search population when they have been verified. As such, the search methods can only evolve as fast as results are validated. Typically, in distributed systems while there are incorrect and malicious results that need to be handled, typically they comprise of a very small portion of all the results in the system. For example, the MilkyWay@Home project reports less than 1% of the results received as invalid.

It may be possible to further improve the time to solution and further reduce the amount of resources dedicated to validation by using an optimistic strategy. Instead of keeping a queue of individuals which require validation, all reported results can be inserted into the search population, and validated individuals can be placed in a backup population. In this way new individuals can be generated from the most up to date population, although this population may contain a few incorrect members. When a member of the search population is found to be invalid, it can be replaced with a previously validated individual from the backup population. This should reduce the amount of resources required to handle validation and allow the search to progress without waiting for any individuals to be validated. The question remains if the possible invalid members in the search population harm the population enough to make these benefits not worthwhile.

CHAPTER 10
Discussion

This thesis describes the implementation of a framework for generic distributed optimization (FGDO). This framework allows optimization problems, optimization methods and different distributed computing platforms to be researched and developed independently and used interoperably through simple interfaces. This framework has been used to perform distributed optimization on various massive scale computing environments, such as RPI's CCNI BlueGene/L and the Milky-Way@Home volunteer computing project. Due to the interoperable nature of this framework, it was possible to examine different optimization strategies, implement a simulation environment to test these strategies on representative benchmark optimization problems with controllable computing environments, and implement a strategy to reduce the amount of result validation required by computing projects which utilize unreliable hosts, such as MilkyWay@Home.

Traditional optimization methods, such as differential evolution, genetic search and particle swarm optimization are iterative, or synchronous in nature. They generate a population of potential solutions (or individuals), and based on that population generate a new population to be evaluated. This limits their scalability to size of these populations. This work introduces asynchronous optimization, a novel approach which separates the scalability of an optimization method from its population size. Additionally, asynchronous optimization does not have any dependencies between evaluated individuals. In traditional parallel optimization methods, if there is a fault with one member of the population, it must be recalculated which slows down the evolution of the populations. If the calculation of the optimization function is nondeterministic, or the computation environment is heterogeneous, the search has to wait for the slowest processor and individual to complete calculation before progressing. Asynchronous optimization can generate any number of new individuals based on its currently known population, and only inserts results to that population when and if they are reported. In this way asynchronous optimization

is naturally scalable, fault tolerant and load balanced, as it does not need to wait for slow or unresponsive hosts.

Differential evolution, genetic search and particle swarm optimization were all modified to asynchronously generate individuals and update their populations using this strategy. Results show that asynchronous versions of these optimizations can scale to hundreds of thousands of computing hosts, while the traditional synchronous versions do not benefit from additional computing hosts as the larger population sizes required reduce their ability to quickly find a solution. In some cases, synchronous optimization even decreases in time to solution as more computing hosts are added. Additionally, the asynchronous versions are largely unaffected by increasing heterogeneity in the computing system, in some cases requiring less results to reach a solution. Even on a complex simulated computing environment modeled from the MilkyWay@Home computing project, the asynchronous searches were shown to scale well.

The validation strategy implemented allows the amount of resources devoted to validation of results to be modified. The effect of this validation rate was examined on the MilkyWay@Home computing project using asynchronous differential evolution, genetic search and particle swarm optimization with various search parameters. As opposed to the default implementation of validation in the BOINC volunteer computing framework, which requires every result to be validated against at least one other duplicated computation, the validation strategy here shows that the amount of redundant computation can be reduced to 30% and potentially less of the results calculated for asynchronous differential evolution and particle swarm. Asynchronous genetic search performed better with a larger amount of computation devoted to validation, around 60%.

This works shows that as computing systems continue to increase in size and heterogeneity, traditional optimization methods need to evolve to deal with new challenges in scalability, heterogeneity and fault tolerance. The asynchronous optimization methods and validation strategies presented here provide strong preliminary work for scalable, fault tolerant optimization strategies that can easily be used on heterogeneous environments. The simulation environment implemented will al-

low for future detailed analysis of the effects of heterogeneity, scale and faults to further improve optimization for massive scale computing systems. Finally, this work has enabled over 25,000 computing hosts to further scientific research in astroinformatics with the MilkyWay@Home volunteer computing project.

LITERATURE CITED

[1] J. et al Adelman-McCarthy. The 6th Sloan Digital Sky Survey Data Release, http://www.sdss.org/dr6/, July 2007. ApJS, in press, arXiv/0707.3413.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[3] Enrique Alba and Bernabe Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 9:126–142, April 2005.

[4] Enrique Alba and Jose M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17:451–465, January 2001.

[5] David P. Anderson, Jeff Cobb, Eric Korpela, Matt Lebofsky, and Dan Werthimer. SETI@home: an experiment in public-resource computing. *Commun. ACM*, 45(11):56–61, 2002.

[6] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203. IEEE Computer Society, 2005.

[7] S. Baskar, A. Alphones, and P. N. Suganthan. Concurrent PSO and FDR-PSO based reconfigurable phase-differentiated antenna array design. In *Congress on Evolutionary Computation*, volume 2, pages 2173–2179, June 2004.

[8] S. Baskar and P. N. Suganthan. A novel concurrent particle swarm optimization. In *Congress on Evolutionary Computation*, volume 1, pages 792–796, June 2004.

[9] F. Van Den Bergh and A. P. Engelbrecht. A cooperative approach to particle swarm optimization. *IEEE Transactions on Evolutionary Computation*, 8(3):225–239, June 2004.

[10] Johan Berntsson and Maolin Tang. A convergence model for asynchronous parallel genetic algorithms. In *IEEE Congress on Evolutionary Computation (CEC2003)*, volume 4, pages 2627–2634, December 2003.

[11] R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, New Mexico, November 1994.

[12] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/G: An architecture for a resource management and scheduling system in a global computational grid. In *4th International Conference on High Performance Computing in the Asia-Pacific Region (HPC Asia 2000)*, pages 283–289, Beijing, China, May 2000.

[13] Erick Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.

[14] Rachid Chelouah and P. Siarry. A continuous genetic algorithm designed for the global optimization of multimodal functions. *Journal of Heuristics*, 6:191–213, 2000.

[15] Rachid Chelouah and Patrick Siarry. Tabu search applied to global optimization. *European Journal of Operational Research*, 123(2):256–270, June 2000.

[16] Rachid Chelouah and Patrick Siarry. Genetic and Nelder-Mead algorithms hybridized for a more accurate global optimization of continuous multiminima functions. *European Journal of Operational Research*, 148(2):335–348, July 2003.

[17] Rachid Chelouah and Patrick Siarry. A hybrid method combining continuous tabu search and Nelder-Mead simplex algorithms for the global optimization of multiminima functions. *European Journal of Operational Research*, 161(3):636–654, March 2005.

[18] M. Clerc. The swarm and the queen: towards a deterministic and adaptive particle swarm optimization. In *Proceedings of the 1999 Congress on Evolutionary Computation*, volume 3, pages 1951–1957, July 1999.

[19] Nathan Cole. *Maximum Likelihood Fitting of Tidal Streams with Application to the Sagittarius Dwarf Tidal Tails*. PhD thesis, Rensselaer Polytechnic Institute, 2009.

[20] Nathan Cole, Heidi Newberg, Malik Magdon-Ismail, Travis Desell, Kristopher Dawsey, Warren Hayashi, Jonathan Purnell, Boleslaw Szymanski, Carlos A. Varela, Benjamin Willett, and James Wisniewski. Maximum likelihood fitting of tidal streams with application to the sagittarius dwarf tidal tails. *Astrophysical Journal*, 683:750–766, 2008.

[21] Xiaohui Cui and Thomas E. Potok. Distributed adaptive particle swarm optimizer in dynamic environment. In *International Parallel and Distributed Processing Symposium*, pages 1–7, March 2007.

[22] Yi Da and Ge Xiurun. An improved PSO-based ANN with simulated annealing technique. In *New Aspects in Neurocomputing: 11th European*

Symposium on Artificial Neural Networks, volume 63, pages 527–533, January 2005.

[23] Sanjoy Das, Praveen Koduru, Min Gui, Michael Cochran, Austin Wareing, Stephen M. Welch, and Bruce R. Babin. Adding local search to particle swarm optimization. In *IEEE Congress on Evolutionary Computation*, pages 428–433, July 2006.

[24] T. Desell, K. El Maghraoui, and C. Varela. Load balancing of autonomous actors over dynamic networks. In *Proceedings of the Hawaii International Conference on System Sciences, HICSS-37 Software Technology Track*, pages 1–10, January 2004.

[25] Travis Desell. Autonomic grid computing using malleability and migration: An actor-oriented software framework. Master's thesis, Rensselaer Polytechnic Institute, May 2007.

[26] Travis Desell, Nathan Cole, Malik Magdon-Ismail, Heidi Newberg, Boleslaw Szymanski, and Carlos Varela. Distributed and generic maximum likelihood evaluation. In *3rd IEEE International Conference on e-Science and Grid Computing (eScience2007)*, pages 337–344, Bangalore, India, December 2007.

[27] Travis Desell, Kaoutar El Maghraoui, and Carlos Varela. Malleable Components for Scalable High Performance Computing . In *Proceedings of the HPDC'15 Workshop on HPC Grid programming Environments and Components (HPC-GECO/CompFrame)*, pages 37–44, Paris, France, June 2006. IEEE Computer Society.

[28] Travis Desell, Boleslaw Szymanski, and Carlos Varela. Asynchronous genetic search for scientific modeling on large-scale heterogeneous environments. In *17th International Heterogeneity in Computing Workshop*, Miami, Florida, April 2008.

[29] Travis Desell, Boleslaw Szymanski, and Carlos Varela. An asynchronous hybrid genetic-simplex search for modeling the milky way galaxy using volunteer computing. In *Genetic and Evolutionary Computation Conference*, Atlanta, Georgia, July 2008.

[30] Travis Desell, Anthony Waters, Malik Magdon-Ismail, Boleslaw Szymanski, Carlos Varela, Matthew Newby, Heidi Newberg, Andreas Przystawik, and Dave Anderson. Accelerating the MilkyWay@Home volunteer computing project with GPUs. *8th International Conference on Parallel Processing and Applied Mathematics (PPAM 2009)*, September 2009.

[31] Zhang Dingxue, Guan Zhihong, and Liu Xinzhi. An adaptive particle swarm optimization algorithm and simulation. In *IEEE International Conference on Automation and Logistics*, pages 2399–2042, August 2007.

[32] Bernabe Dorronsoro and Enrique Alba. A simple cellular genetic algorithm for continuous optimization. *IEEE Congress on Evolutionary Computation (CEC2006)*, pages 2838–2844, July 2006.

[33] Bernabe Dorronsoro, Enrique Alba, Mario Giacobini, and Marco Tomassini. The influence of grid shape and asynchronicity on cellular evolutionary algorithms. In *IEEE Congress on Evolutionary Computation (CEC2004)*, volume 2, pages 2152–2158, June 2004.

[34] R. C. Eberhart and J. Kennedy. A new optimizer using particle swarm theory. In *Sixth International Symposium on Micromachine and Human Science*, pages 33–43, 1995.

[35] R. C. Eberhart and Y. Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the 2000 Congress on Evolutionary Computation*, volume 1, pages 84–88, July 2000.

[36] Wei Fang, Jun Sun, Wenbo Xu, and Jing Liu. Fir digital filters design based on quantum-behaved particle swarm optimization. In *First International Conference on Innovative Computing, Information and Control*, volume 1, pages 615–619, 2006.

[37] Gianluigi Folino, Agostino Forestiero, and Giandomenico Spezzano. A JXTA based asynchronous peer-to-peer implementation of genetic programming. *Journal of Software*, 1:12–23, August 2006.

[38] I. Foster and C. Kesselman. The Globus Project: A Status Report. In J. Antonio, editor, *Proceedings of the Seventh Heterogeneous Computing Workshop (HCW '98)*, pages 4–18. IEEE Computer Society, March 1998.

[39] I. Foster and C. Kesselman. Globus: A toolkit-based grid architecture. In *The Grid: Blueprint for a New Computing Infrastructure*, pages 259–278. Morgan Kaufmann, 1999.

[40] Francesco Franz and Nicol Speciale. A tabu-search-based algorithm for continuous multiminima problems. *International Journal for Numerical Methods in Engineering*, 50(3):507–759, January 2001.

[41] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. *Cluster Computing*, 5(3):237–246, 2002.

[42] Hongwei Ge, Wenli Du, and Feng Qian. A hybrid algorithm based on particle swarm optimization and simulated annealing for job shop scheduling. In *Third International Conference on Natural Computation*, volume 3, pages 715–719, August 2007.

[43] Li Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5:88–95, May/June 2001.

[44] Wenyin Gong and Zhihua Cai. Differential evolution made faster and more robust. In *IEEE International Conference on Industrial Technology 2006 (ICIT2006)*, pages 606–611, Mumbai, December 2006.

[45] A. S. Grimshaw, M. A. Humphrey, and A. Natrajan. A philosophical and technical comparison of Legion and Globus. *IBM J. Res. Dev.*, 48(2):233–254, 2004.

[46] Hiroaki Imade, Ryohei Morishita, Isao Ono, Norihiko Ono, and Masahiro Okamoto. A grid-oriented genetic algorithm framework for bioinformatics. *New Generation Computing: Grid Systems for Life Sciences*, 22:177–186, January 2004.

[47] Chia-Feng Juang. A hybrid of genetic algorithm and particle swarm optimization for recurrent network design. *IEE Transactions on Systems, Man and Cybernetics, Part B*, 34(2):997–1006, April 2004.

[48] Venkatesh Katari, Suresh Chandra Satapathy, JVR Murthy, and PVGD Prasad Reddy. Hybridized improved genetic algorithm with variable length chromosome for image clustering. *IJCSNS International Journal of COmputer Science and Network Security*, 7(11):121–131, November 2007.

[49] J. Kennedy and R. C. Eberhart. Particle swarm optimization. In *IEEE International Conference on Neural Networks*, volume 4, pages 1942–1948, 1995.

[50] Kryzysztof C. Kiwiel. Proximity control in bundle methods for convex nondifferentiable minimization. *Mathematical Programming*, 46(1):105–122, May 2005.

[51] Praveen Koduru, Sanjoy Das, and Stephen Welch. A particle swarm optimization-Nelder Mead hybrid algorithm for balanced exploration and exploitation in multidimensional search space. In Hamid R. Arabnia, editor, *IC-AI*, pages 457–464. CSREA Press CSREA Press, 2006.

[52] Byung-Il Koh, Alan D. George, and Raphael T. Haftka. Parallel asynchronous particle swarm optimization. *International Journal of Numerical Methods in Engineering*, 67(4):578–595, July 2006.

[53] Andrew Lewis and David Abramson. An evolutionary programming algorithm for multi-objective optimisation. In *IEEE Congress on Evolutionary Computation (CEC2003)*, volume 3, pages 1926–1932, December 2003.

[54] J. J. Liang, A. K. Qin, P. M. Suganthan, and S. Baskar. Particle swarm optimization with novel learning strategies. In *IEEE International Conference on Systems, Man and Cybernetics*, volume 4, pages 3659–3664, October 2004.

[55] Dudy Lim, Yew-Soon Ong, Yaochu Jin, Bernhard Sendhoff, and Bu-Sung Lee. Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23:658–670, May 2007.

[56] Jing Liu and Jun Sun. Quantum-based particle swarm optimization based on immune memory and vaccination. In *IEEE International Conference on Granular Computing*, pages 453–456, May 2006.

[57] Jing Liu, Wenbo Xu, and Jun Sun. Quantum-behaved particle swarm optimization with mutation operator. In *International Conference on Tools with Artificial Intelligence*, November 2005.

[58] K. El Maghraoui, J. Flaherty, B. Szymanski, J. Teresco, and C. Varela. Adaptive computation over dynamic and heterogeneous networks. In R. Wyrzykowski, J. Dongarra, M. Paprzycki, and J. Wasniewski, editors, *Proc. of the Fifth International Conference on Parallel Processing and Applied Mathematics (PPAM'2003)*, number 3019 in LNCS, pages 1083–1090, Czestochowa, Poland, September 2003.

[59] Kaoutar El Maghraoui, Travis J. Desell, Boleslaw K. Szymanski, and Carlos A. Varela. The internet operating system: Middleware for adaptive distributed computing. *International Journal of High Performance Computing Applications (IJHPCA), Special Issue on Scheduling Techniques for Large-Scale Distributed Platforms*, 10(4):467–480, 2006.

[60] Message Passing Interface Forum. MPI: A message-passing interface standard. *The International Journal of Supercomputer Applications and High Performance Computing*, 8(3/4):159–416, Fall/Winter 1994.

[61] E. Mezura-Montes, J. Velazquez-Reyes, and C.A C. A. Coello Coello. Modified differential evolution for constrained optimization. In *IEEE Congress on Evolutionary Computation 2006 (CEC2006)*, pages 25–32, Vancouver, BC, July 2006.

[62] Efrn Mezura-Montes, Jess Velzquez-Reyes, and Carlos A. Coello Coello. A comparative study of differential evolution variants for global optimization. In *Proceedings of the 8th Annual Conference on Genetic and Evolutionary Computation*, pages 485–492, 2006.

[63] Kalsa Miettinen, Marko M. Mkel, and Heikki Maaranen. Efficient hybrid methods for global continuous optimization based on simulated annealing.

Computers and Operations Research, 33(4):1102–1116, April 2006. Part Special Issue: Optimization Days.

[64] A. Natrajan, A. Nguyen-Tuong, M. Humphrey, and A. Grimshaw. The Legion grid portal. *Grid Computing Environments Special Issue, Concurrency and Computation: Practice and Experience*, 14(13-14):1365–1394, 2002.

[65] Vijay Pande et al. Atomistic protein folding simulations on the submillisecond timescale using worldwide distributed computing. *Biopolymers*, 68(1):91–109, 2002. Peter Kollman Memorial Issue.

[66] K. E. Parsopoulos, D. K. Tasoulis, N. G. Pavlidis, V. P. Plagianakos, and M. N. Vrahatis. Vector evaluated differential evolution for multi-objective optimization. In *Congress on Evolutionary Computation 2004 (CEC2004)*, volume 1, pages 204–211, June 2004.

[67] T. Peachey, D. Abramson, and A. Lewis. Model optimization and parameter estimation with Nimrod/o. In *International Conference on Computational Science*, University of Reading, UK, May 2006.

[68] T. Peram, K. Veeramachaneni, and C. K. Mohan. Fitness-distance-ratio based particle swarm optimization. In *Proceedings of the IEEE Swarm Intelligence Symposium*, pages 174–181, 2003.

[69] Y. G. Petalas, K. E. Parsopoulos, and M. N. Vrahatis. Entropy-based memetic particle swarm optimization for computing periodic orbits of nonlinear mappings. In *IEEE Congress on Evolutionary Computation*, pages 2040–2047, September 2007.

[70] Mitchell A. Potter and Kenneth De Jong. A cooperative coevolutionary approach to function optimization. In Yuval Davidor, Hans-Paul Schwefel, and Reinhard Männer, editors, *Parallel Problem Solving from Nature – PPSN III*, pages 249–257, Berlin, 1994. Springer.

[71] Jonathan Purnell, Malik Magdon-Ismail, and Heidi Newberg. A probabilistic approach to finding geometric objects in spatial datasets of the Milky Way. In *Proceedings of the 15th International Symposium on Methodoligies for Intelligent Systems (ISMIS 2005)*, pages 475–484, Saratoga Springs, NY, USA, May 2005. Springer.

[72] J. R. Prez and J. Basterrechea. Particle swarms applied to array synthesis and planar near-field antenna measurements. *Microwave and Optical Technology Letters*, 50(2):544–548, February 2008.

[73] S. Rahnamayan, H. R. Tizhoosh, and M. M. A. Salama. Opposition-based differential evolution algorithms. In *IEEE Congress on Evolutionary Computation 2006 (CEC2006)*, pages 2010–2016, Vancouver, BC, July 2006.

[74] Jean-Michel Renders and Hugues Bersini. Hibridizing genetic algorithms with hill-climbing methods for global optimization: two possible ways. In *IEEE World Congress on Computational Intelligence, First Conference on Evolutionary Computation*, volume 1, pages 312–317, June 1994.

[75] Suresh Chandra Satapathy, JVR Murthy, PVGD Prasada, Venkatesh Katari, Satish Malireddi, and VNK Srujan Kollisetty. An efficient hybrid algorithm for data clustering using improved genetic algorithm and nelder mead simplex search. In *International Conference on Computational Intelligence and Multimedia Applications*, volume 1, pages 498–510, December 2007.

[76] J. F. Schutte, J. A. Reinbolt, B. J. Fregly, R. T. Haftka, and A. D. George. Parallel global optimization with the particle swarm algorithm. *International Journal for Numerical Methods in Engineering*, 61(13):2296–2315, December 2004.

[77] H.-P. Schwefel. *Evolution and Optimization Seeking*. John Wiley & Sons, New York, 1995.

[78] Gregory Seront and Hugues Bersini. Simplex GA and hybrid methods. In *IEEE International Conference on Evolutionary Computation*, pages 845–848, May 1996.

[79] C. E. Shannon. *The Mathematical Theory of Communication*. University of Illinios Press, 1964.

[80] Y. Shi and R. C. Eberhart. A modified particle swarm optimizer. In *IEEE World Congress on Computational Intelligence*, pages 69–73, May 1998.

[81] P. Siarry and G. Berthiau. Fitting of tabu search to optimize functions of continuous variables. *International Journal for Numerical Methods in Engineering*, 40(13):2249–2457, September 1997.

[82] Abhishek Sinha and David E. Goldberg. A survey of hybrid genetic and evolutionary algorithms. Technical Report No. 2003004, Illinois Genetic Algorithms Laboratory (IlliGAL), 2003.

[83] F. Solis and R. Wets. Minimization by random search techniques. *Mathematics of Operations Research*, 6:19–30, 1981.

[84] R. Storn and K. Price. Minimizing the real functions of the ICEC'96 contest by differential evolution. In *Proceedings of the IEEE International Conference on Evolutionary Computation*, pages 842–844, Nagoya, Japan, 1996.

[85] Jun Sun, Wenbo Xu, and Bin Feng. A global search strategy of quantum-behaved particle swarm optimization. In *IEEE Conference on Cybernetics and Intelligent Systems*, pages 111–116, December 2004.

[86] Jun Sun, Wenbu Xu, and Bin Feng. Particle swarm optimization with particles having quantum behavior. In *Congress on Evolutionary Computation*, volume 1, pages 325–331, June 2004.

[87] Boleslaw Szymanski, Travis Desell, and Carlos Varela. The effect of heterogeneity on asynchronous panmictic genetic search. In *Proc. of the Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM'2007)*, LNCS, Gdansk, Poland, September 2007.

[88] D. K. Tasoulis, N. G. Pavlidis, V. P. Plagianakos, and M. N. Vrahatis. Parallel differential evolution. In *Congress on Evolutionary Computation 2004 (CEC2004)*, volume 2, pages 2023–2029, June 2004.

[89] Carlos Varela and Gul Agha. Programming dynamically reconfigurable open systems with SALSA. *ACM SIGPLAN Notices. OOPSLA'2001 Intriguing Technology Track Proceedings*, 36(12):20–34, December 2001. http://www.cs.rpi.edu/~cvarela/oopsla2001.pdf.

[90] K. Veeramachaneni, T. Peram, C. K. Mohan, and L. A. Osadciw. Optimization using particle swarms with near neighbor interactions. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 110–121, 2003.

[91] Gerhard Venter and Jaroslaw Sobieszczanski-Sobieski. A parallel particle swarm optimization algorithm accelerated by asynchronous evaluations. In *Sixth World Congresses of Structural and Multidisciplinary Optimization*, pages 1–10, May 2005.

[92] J. Vesterstrom and R. Thomsen. A comparative study of differential evolution, particle swarm optimization, and evolutionary algorithms on numerical benchmark problems. In *Congress on Evolutionary Computation 2004 (CEC2004)*, volume 2, pages 1980–1987, June 2004.

[93] Mingxing Wang, Xi Chen, and Jixin Qian. An improvement of continuous tabu search for global optimization. In *Fifth World Congress on Intelligent Control and Automation*, volume 1, pages 375–377, June 2004.

[94] Xue Wang, Jun-Jie Ma, Sheng Wang, and Dao-Wei Bi. Distributed particle swarm optimization and simulated annealing for energy-efficient coverage in wireless sensor networks. *Sensors Special Issue on Energy Efficiency and Intelligent Signal Processing for Wireless Sensing*, 7(5):628–648, May 2007.

[95] Z. G. Wang, M. Rahman, Y. S. Wong, and J. Sun. Optimization of multi-pass milling using parallel genetic algorithm and parallel genetic simulated annealing. *International Journal of Machine Tools and Manufacture*, 45(15):1726–1734, December 2005.

[96] Z. G. Wang, Y. S. Wong, and M. Rahman. Development of a parallel optimization method based on genetic simulated annealing algorithm. *Parallel Computing*, 31(8–9):839–857, August–September 2005.

[97] Lingyun Wei and Mei Zhao. A niche hybrid genetic algorithm for global optimization of continuous multimodal functions. *Applied Mathematics and Computation*, 160(3):649–661, January 2005.

[98] Lei Xu and Fengming Zhang. Parallel particle swarm optimization for attribute reduction. In *Eighth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, volume 1, pages 770–775, July 2007.

[99] Jinn-Yi Yeh and J. C. Fu. Parallel adaptive simulated annealing for computer-aided measurement in functional MRI analysis. *Expert Systems with Applications*, 33(3):706–715, October 2007.

[100] J. Yen, J. C. Liao, Bogju Lee, and D. Randolph. A hybrid approach to modeling metabolic systems using a genetic algorithm and simplex method. *IEEE Transactions on Systems, Man and Cybernetics, Part B*, 29(2):173–191, April 1998.

[101] Wen-Jun Zhang and Xiao-Feng Xie. Depso: hybrid particle starm with differential evolution operator. In *IEEE International Conference on Systems, Man and Cybernetics 2003*, volume 4, pages 3816–3821, October 2003.