

# Asynchronous Genetic Search for Scientific Modeling on Large-Scale Heterogeneous Environments

Travis Desell

Boleslaw Szymanski

Carlos Varela

Department of Computer Science  
Rensselaer Polytechnic Institute, Troy, NY, U.S.A.  
{deselt, szymansk, cvarela}@cs.rpi.edu

## Abstract

*Use of large-scale heterogeneous computing environments such as computational grids and the Internet has become of high interest to scientific researchers. This is because the increasing complexity of their scientific models and data sets is drastically outpacing the increases in processor speed while the cost of supercomputing environments remains relatively high. However, the heterogeneity and unreliability of these environments, especially the Internet, make scalable and fault tolerant search methods indispensable to effective scientific model verification. The paper introduces two versions of asynchronous master-worker genetic search and evaluates their convergence and performance rates in comparison to traditional synchronous genetic search on both a IBM BlueGene supercomputer and using the MilkyWay@HOME BOINC Internet computing project<sup>1</sup>. The asynchronous searches not only perform faster on heterogeneous grid environments as compared to synchronous search, but also achieve better convergence rates for the astronomy model used as the driving application, providing a strong argument for their use on grid computing environments and by the MilkyWay@Home BOINC internet computing project.*

## 1 Introduction

The rate of increase in CPU performance does not nearly match the rapidly increasing rates of data acquisition in all scientific disciplines. This is leading to significantly long, if not intractable, turn around times between the development of a scientific model and its verification using traditional computing environments. Testing current scientific models can involve processing terabytes of data using computationally intense modeling techniques, which results in program execution times of weeks to months on a single high end computer to calculate only a single parameter set for a scientific model. Even the best scientific model verification search methods require the evaluation of thousands of parameter sets of a scientific model over the data set. This makes using large-scale computing environments, such as computational grids and the Internet, highly desirable platforms for performing scientific model verification. The processing power these environments provide enables them to compute these models in a short amount of time, which in turn allows scientists to more quickly gather results and improve their models and understanding.

Grids and the Internet involve additional challenges in comparison to homogeneous large-scale computing environments such as supercomputers. In addition to scalability and heterogeneity concerns, the reliability of the host nodes comes into question, especially in the case of internet computing architectures such as BOINC [4] where

---

<sup>1</sup>See <http://milkyway.cs.rpi.edu>

computing nodes can disconnect at random and for computationally significant amounts of time. Most search methods used in scientific model verification are iterative (or synchronous) in nature [8], and are not well suited to heterogeneous and unreliable computing environments. With iterative searches, new parameter sets for evaluation depend on either the previous or a group of previous evaluations having been completed, and the search will only progress as fast as the slowest results are received. Traditional redundancy techniques used for fault-tolerance can result in either lack of progress in the iterative computation for checkpointing and restart, or significant amounts of wasted CPU cycles for redundancy. Iterative methods also tend not to scale well and can be very difficult to partition over heterogeneous resources.

This work examines the feasibility of using asynchronous genetic search for large-scale computing environments such as BOINC, which are highly unstable and heterogeneous, but can potentially offer millions of processors [4]. A software framework for distributed scientific model evaluation (GMLE [9]) is extended by providing support for asynchronous search methods. Two asynchronous variations of the genetic search algorithm are introduced and their convergence times are compared with traditional genetic search by using an astronomical modeling application [19]. The effect of asynchronicity and heterogeneity on the performance of the searches is shown by using the MilkyWay@HOME BOINC computing project. The potential increase in performance obtained by using these asynchronous algorithms is demonstrated using a heterogeneous grid environment.

## 2 Related Work

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) [8]. Typically, these ap-

proaches are synchronous. Panmictic GAs create a population, evaluate it in parallel, and use the results to generate the next population. Island [3, 5] approaches evaluate local populations for a certain number of iterations, then exchange the best members with other islands. Cellular algorithms [2, 10] evaluate individual parameter sets, then update these individual sets based on the fitness of their neighbors. Hybrid approaches [16, 20] have also been examined.

P-CAGE [12] is a peer-to-peer (P2P) implementation of a hybrid multi-island genetic search built using the JXTA protocol [14] 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 [15], using the Globus Toolkit [13]. Lim et. al. provide a framework for distributed calculation of genetic algorithms and an extended API and meta-scheduler for resource discovery [17]. Both approaches use synchronous island-style GAs. Nimrod/O [18] is a tool that provides different optimization algorithms for use with Nimrod/G [7]. Nimrod/O has been used to develop the EPSOC algorithm [16] which 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.

It has already been shown by Dorronsoro et. al. that asynchronous cellular GAs can perform competitively and discuss how update rate and different population shapes affect the convergence rate [11]. In this paper, we introduce a novel approach (to the best of our knowledge) that evaluates asynchronous panmictic GAs. This approach is well suited for internet computing using the BOINC infrastructure, because it easily facilitates scalability, fault tolerance without redundancy, and does not require inter-worker communication

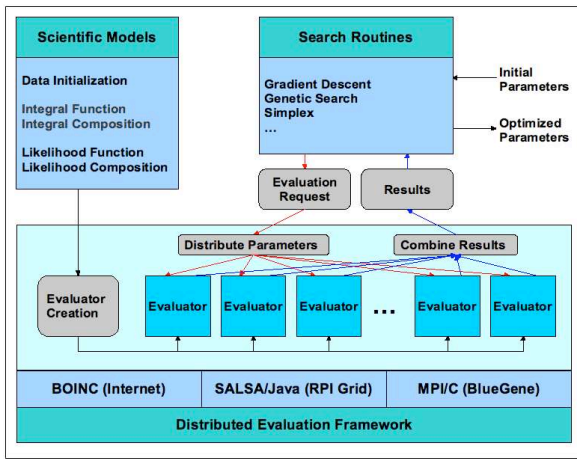


Figure 1. GMLE with a synchronous distributed evaluation framework.

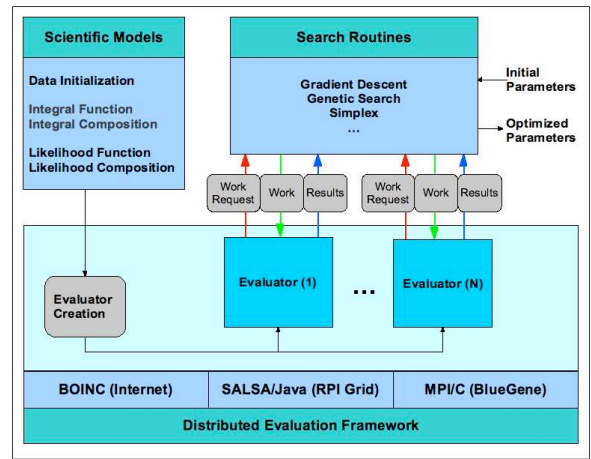


Figure 2. GMLE with an asynchronous distributed evaluation framework.

(which is prohibited by BOINC).

### 3 Optimization on Heterogeneous Environments

The GMLE framework has been designed to facilitate collaboration between researchers in machine learning, distributed computing and experts with different scientific domain knowledge who are interested in distributed model verification or parameter optimization. GMLE previously has used a synchronous distributed evaluation framework (see Figure 1) for performing maximum likelihood evaluation on astronomical and particle physics applications on an IBM BlueGene supercomputer and the Rensselaer Grid [9]. The framework partitions data across a set of processors which perform partial evaluations of the model in parallel, after which the results are composed into the final result. This has been shown to be efficient for both supercomputing and grid environments, however it does not work well on highly heterogeneous and unstable environments like the BOINC infrastructure and some grids.

GMLE was extended with an asynchronous distributed evaluation framework (see Figure 2). Evaluators request work from a master, process that work and return the result, repeating as nec-

essary. Work requests and results are all processed asynchronously by the master which performs the different search methods. The master does not need to wait or have any dependencies on the results of the different evaluators which makes evaluator failures easily ignored and reduces the need for redundant, wasted computations.

### 4 Search Methods

**Iterative Genetic Search** Algorithm 1 shows pseudocode for the IGS algorithm. In this algorithm, an initial population of parameter sets is generated randomly and the fitness of the model for each of those parameter sets is calculated. The iterative genetic search repeatedly calculates a new population based on the previous one using selection, reproduction and mutation. Selection takes the best members of the previous population and moves them to the new population. Reproduction takes two randomly selected members of the previous population and generates a new parameter set that is their average. Mutation takes a randomly selected member of the previous population and creates a new parameter set which is equal to the selected member except that one value is mutated to a new randomly selected value. In this way, iterative genetic search will converge

---

**Algorithm 1: Iterative Genetic Search (IGS)**

---

**Data:**  $X$  /\*Best to keep\*/,  $Y$  /\*Number Reproductions\*/,  $Z$  /\*Number Mutations\*/

**Result:** Converged Population

**for**  $p \in P[1] \dots P[X+Y+Z]$  **do**  
└  $p.params = random\_params()$

evaluate( $P$ )

**while** *not converged*( $P$ ) **do**

┌ **for**  $p \in P'[1] \dots P'[X]$  **do**  
└  $p = P.get\_next\_best()$

┌ **for**  $p \in P'[X+1] \dots P'[X+Y]$  **do**  
└  $p = reproduce(P[random()], P[random()])$

┌ **for**  $p \in P'[X+Y+1] \dots P'[X+Y+Z]$  **do**  
└  $p = mutate(P[random()])$

$P = P'$

evaluate( $P$ )

---

to minima using reproduction and use mutation to prevent being stuck in a local minimum. The population size,  $S$ , is typically kept constant, so  $S = X + Y + Z$ , where  $X$  is the number of selections,  $Y$  is the number of reproductions, and  $Z$  is the number of mutations.

There are three ways that IGS can be parallelized: (1) the fitness of each member in the population can be evaluated in parallel, (2) the fitness calculation can be done in parallel, and (3) the fitness calculation can be done in parallel as well as the population being evaluated in parallel.

The first approach can scale to a number of processors equal to the population size, while the scalability of the second approach is dependent on how much of the fitness calculation can be done in parallel. The third approach can scale to a number of processors equal to the first times the second, however it is the most complex to implement. All three approaches suffer from the scalability limitation imposed either by the population size and/or the scalability of the fitness calculation. None perform well on heterogeneous environments without intelligent partitioning. In the first case, the algorithm will only progress as fast as the slowest fitness calculation, while in the second case, the

---

**Algorithm 2: Asynchronous Report Work**

---

**Data:**  $P$  /\*Population\*/,  $max$  /\*Maximum Population Size\*/,  $R$  /\*Result\*/

**Result:** Updated Population

**if**  $P.size < max$  **then**

└  $P.insert(R)$

**else if**  $R.fitness > worst(P).fitness$  **then**

└  $P.insert(R)$

└  $P.remove(worst(P))$

---

---

**Algorithm 3: Asynchronous Request Work**

---

**Data:**  $P$  /\*Population\*/,  $C$  /\*Reproduction Probability\*/,  $max$  /\*Maximum Population Size\*/

**Result:** New Parameters to Evaluate

**if**  $P.size < max$  **then**

└ return  $random\_params()$ ;

**else**

┌ **if**  $random() < C$  **then**

└  $p1 = P[random()]$

└  $p2 = P[random()],$  where  $p1 \neq p2$

└ return  $reproduce(p1, p2)$

**else**

└ return  $mutate(P[random()])$

---

algorithm will only progress as fast as the slowest calculation of part of the fitness. The third case suffers from both, making partitioning the most difficult.

**Asynchronous Genetic Search** AGS is similar to IGS in that it keeps a population of parameters and generates reproductions and mutations based on it. However, instead of using a parallel model of concurrency like IGS, it uses a master-worker approach. Instead of iteratively generating new populations, new members of the population are generated when a worker requests work, and the population is updated when a worker reports work to the master. The AGS algorithm consists of two phases and uses two asynchronous message handlers (see Algorithms 2 and 3). The actor model of computation [1] is assumed, so the server can either be processing a *request work* or a

*report work* message and cannot process multiple messages at the same time.

In the first phase of the algorithm (while the population size is less than the maximum population size) the server is being initialized and a random population is generated. When a *request work* message is processed, a random parameter set is generated, and when a *report work* message is processed, the population is updated with the parameters and the fitness of that evaluation. When enough *report work* messages have been processed, the algorithm proceeds into the second phase which actually performs the genetic search.

In the second phase, *report work* will insert the new parameters and their fitness into the population but only if they are better than the worst current member, and remove the worst member to keep the population size the same, otherwise the parameters and the result are discarded. Processing a *request work* message will either return a mutation or reproduction from the population.

This algorithm has significant benefits in heterogeneous environments because the calculation of fitness can be done by each worker concurrently and independently of each other. The algorithm progresses as fast as work is received, and faster workers can process multiple *request work* messages, in the style of CILK’s work stealing [6], without waiting on slow workers. Additionally, slower workers still improve the speed of the algorithm because their results can still be useful no matter when they are received. The only limiting factor to scalability is how fast results can be inserted into the population and how fast *request work* messages can be processed. It is also possible to have multiple masters sharing the same population for even greater scalability.

The AGS algorithm was extended with the double shot method, on the observation that for the astronomy model (along with many other scientific modeling applications), the parameter space is not well formed. In this case, when a reproduction is generated from two parameter sets, they often both lie on a slope, so using the average of two points will typically would not improve the fitness. The AGS double shot (AGS-DS) algorithm

---

**Algorithm 4:** Double Shot Reproduce

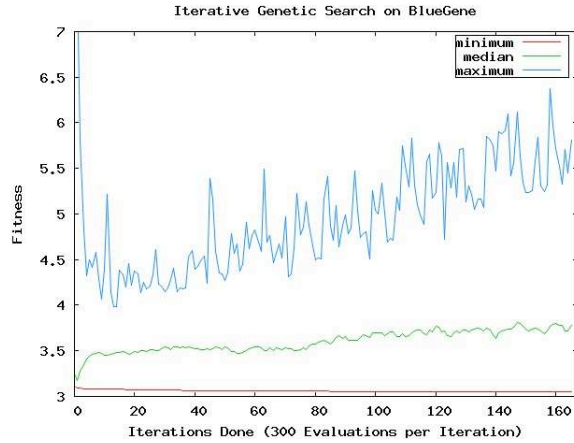
---

```

Data: Member m1, Member m2
Result: Reproduced parameters
Member[] result
result[0].params = (m1.params +
m2.params)/2
diff = result[0].params - m1.params
result[1].params = diff - m1.params
result[2].params = diff + m2.params
return result

```

---

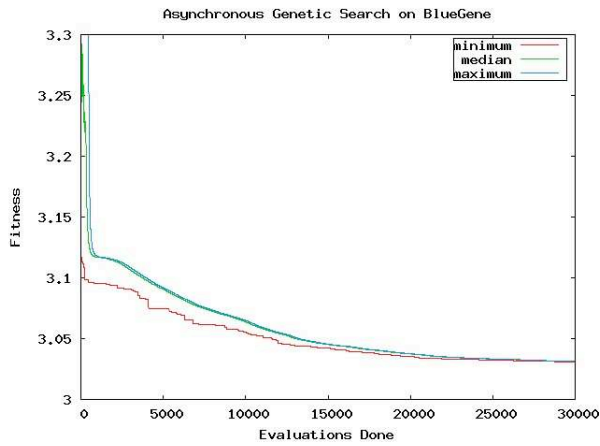


**Figure 3. Minimum, median and maximum values of the iterative genetic search population (updated every 300 evaluations).**

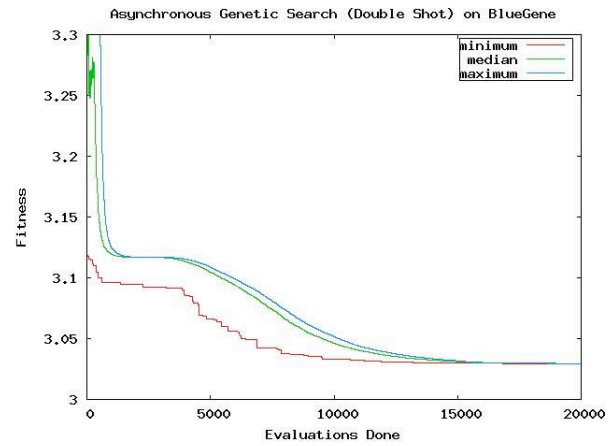
improves AGS by generating three children when doing a reproduction (see Algorithm 4). One child is the average of its parents, but the other two children lie outside the parent parameters. One child is equally distant from the average outside the first parent, and the other child is equally distant from the average outside the second parent. This allows the population to travel down gradients much faster leading to improved convergence times.

## 5 Results

**BlueGene Convergence** The convergence rates of the IGS, AGS and AGS-DS algorithms were tested on Rensselaer’s CCNI BlueGene



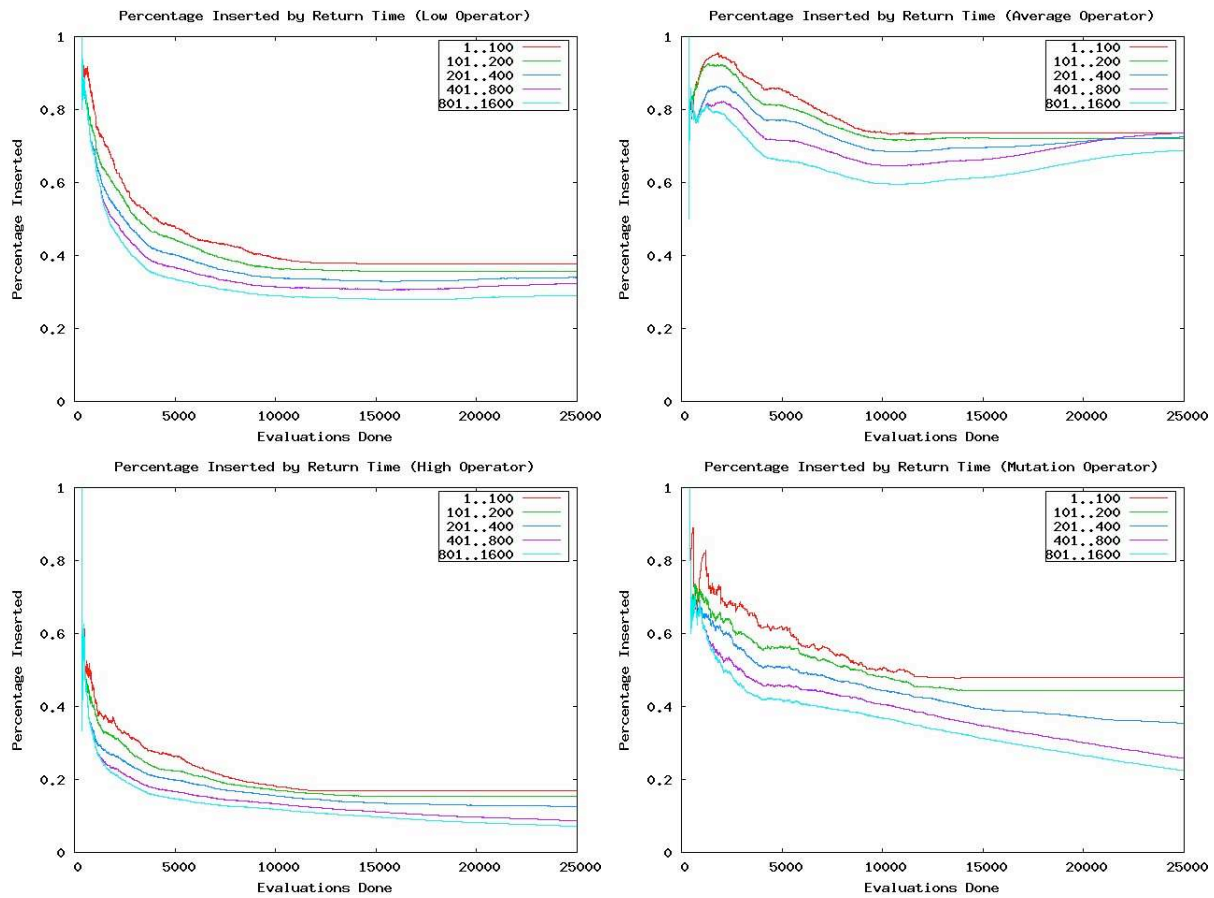
**Figure 4. Minimum, median and maximum values for the asynchronous genetic search population updated every evaluation.**



**Figure 5. Minimum, median and maximum values for the asynchronous double shot genetic search population updated every evaluation.**

because of the expensive fitness calculation – 15 to 30 minutes for a single evaluation on the sample data set running on a high end processor. A 512 node partition was used in virtual mode for a total of 1024 processors, each a 700MHz PowerPC 440 processor with 1GB of RAM connected by a 3-dimensional torus with 175MBps in each direction and  $1.5\mu\text{sec}$  latency. Each search was run with 10 different initial starting populations of size 300 and allowed to run for 50,000 evaluations. AGS and AGS-DS used a reproduction rate of 0.8 and IGS selected the best 30, reproduced 240 and mutated 30 members. Figures 3, 4, and 5 show the convergence rates of IGS, AGS and AGS-DS respectively. They show the average across the different runs of the best member (min), the worst member (max), and the median member for each evaluation. The AGS and AGS-DS algorithms converged within 30,000 evaluations, so only the first 30,000 were shown. The maximum member for IGS fluctuated between 4 and 10 for all iterations of the population. This was due to the fact that the median and maximum included poor mutations and reproductions for IGS, while with AGS and AGS-DS these poorly generated members are never inserted into the population.

The known optimal fitness for the sample data set used was approximately 3.026. IGS still had not converged to the optimum after 50,000 evaluations, while AGS took approximately 30,000 evaluations and AGS-DS took 18,000 evaluations. Both AGS and AGS-DS quickly converged to a local minimum in the data set (at a fitness of approximately 3.1). AGS-DS converged faster to both minima (the local and the optimal) due to the double shot technique allowing the algorithm to travel down gradients quicker. IGS provided the most variation in sampled population points, preventing it from being stuck at any local minima, however this reduced the convergence rate. Another contributing factor could be that in generating populations iteratively, the quality of the newly selected members is on average lower than a continuously updated population, as done in the asynchronous searches. For the astronomical model and the sample data set, the results show that the faster convergence rates of AGS and AGS-DS outperform IGS, with the wider search space of IGS not providing much benefit due to the relatively small number of local minima in the data set.

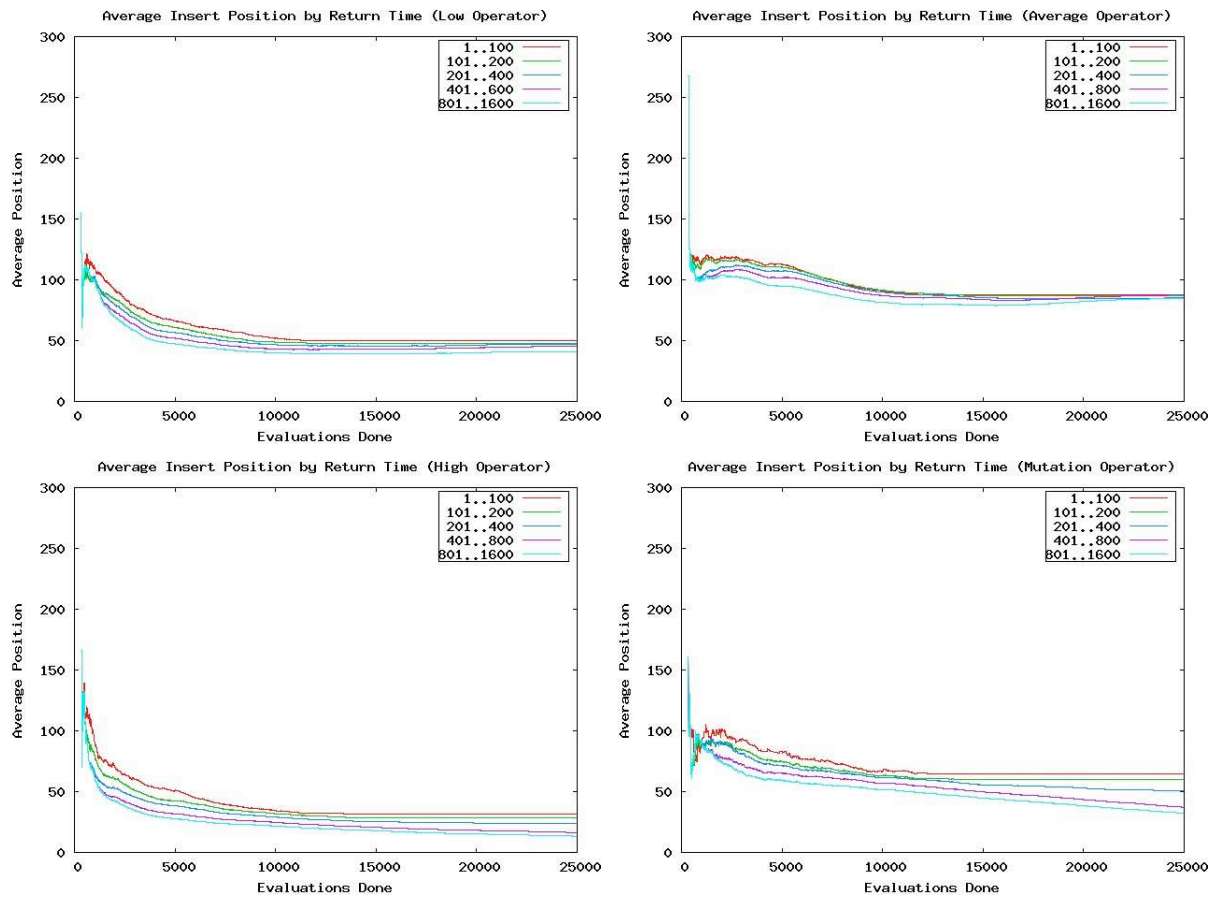


**Figure 7. Percentage of members generated by the different operators inserted into the population as the AGS-DS search progressed on BOINC, based on how many results were reported while the member was being evaluated.**

**BOINC Convergence** The convergence of the AGS-DS search was also evaluated using GMLE and the BOINC Internet computing framework. Figure 6 shows the progression of the average minimum, maximum and median values of five different search populations. AGS-DS converged to the known optimum in 22,000 evaluations, or 1.2 times slower than on the BlueGene. These results are promising, as with the current user base of around 1,000 different processors, the number of evaluations processed per second by BOINC is nearly the same as the 1024 processor partition on the BlueGene, access is not limited and shared with other users and operational costs are significantly less. The decreased performance is due to the fact that

GMLE on the BlueGene would evaluate one member at a time and insert it into the population, so it is always generating new members from the most evolved population. Using BOINC, many different members are being evaluated concurrently so the search is not always using the best possible information, similar to IGS. This does however, improve the global aspects of the search.

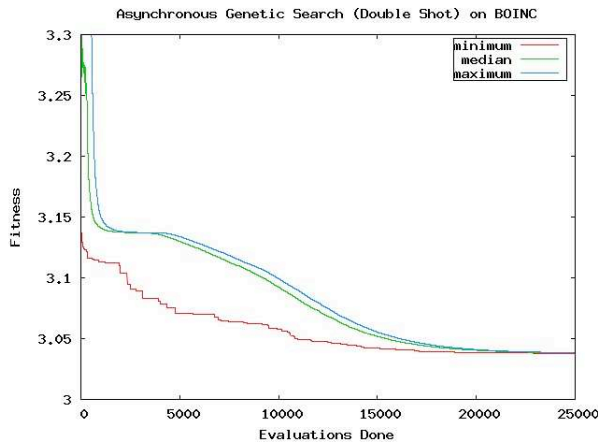
Due to the fact that it could take a very long time for a member to be evaluated and the fitness reported, the utility of the different operations used in the AGS-DS was also measured, in conjunction with how long the members took to be evaluated and the results reported. Figure 7 shows the percentage of members that were in-



**Figure 8. Average position in the population a member is inserted to, based on how many results were reported while the member was being evaluated and how it was generated. Lower positions have better fitness.**

served based on the different operators and how long they took to evaluate. Figure 8 shows what position the members were inserted into the population, with lower positions being more fit with respect to the rest of the population. The *low* operator refers to the member generated by the double shot approach that is on the outside of the more fit parent, and the *high* operator refers to the member generated by the double shot approach that is on the outside of the less fit parent. Interestingly, while adding the low and high operators with the double shot approach improves the convergence rate, the probability that members generated with these operators are inserted into the population is less than with the average message.

However, when they are inserted into the population, they are much more fit than the typical member generated with the average operator. It is also important to note that while members that take longer to be reported are less likely to be inserted into the population, when they are inserted they typically are more fit. This analysis of the position and insert chance suggests that the high and low operators, as well as members reported from slower computers are more explorative benefiting the global search quality of the genetic algorithm, while the average operator and results reported faster tend to be more exploitative focus on local search. These are very promising results for use of asynchronous genetic search on very large scale

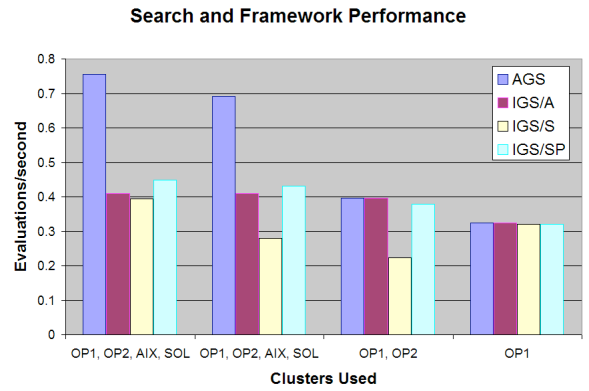


**Figure 6. Minimum, median and maximum values for the asynchronous double shot genetic search population updated every evaluation running on the BOINC framework.**

and heterogeneous networks, because they mean that adding slower processors to the computation still contributes to the search and improves the time to convergence.

**Performance** All three search methods were also tested using the GMLE distributed evaluation framework on four different clusters in the Rensselaer Grid to show how a heterogeneous environment would effect the performance of evaluations. The Solaris cluster (SOL) consists of four single core, dual processor SunBlade 1000 Sun Solaris machines, running at 800MHz. The AIX cluster (AIX) consists of four quad-processor single-core Power-PC processors running at 1.7GHz. Two Opteron clusters were also used. The first (OP1) consists of 8 quad-processor, single-core machines, and the second (OP2) consists of 2 quad-processor, dual-core machines with each core running at 2.2MHz. Inter-cluster communication is over the Rensselaer’s wide-area network (WAN).

The synchronous GMLE framework was tested with IGS using partitioned and non-partitioned data. Additionally, asynchronous GMLE was tested using IGS and AGS. A population size of 50 was used to demonstrate how asynchronous IGS



**Figure 9. Performance of AGS and IGS on the Rensselaer Grid. IGS was used with the asynchronous framework (IGS/A), on the synchronous framework (IGS/S), and on the synchronous framework with partitioned data (IGS/SP).**

can only scale up to the population size. The non-partitioned IGS, asynchronous IGS and AGS used one evaluator per processor, while the partitioned IGS used one evaluator per Solaris processor, two evaluators per AIX processor, and three evaluators per core on the Opteron clusters.

Figure 9 shows the number of evaluations performed per second using different combinations of these clusters. Evaluations were done over a reduced data set. AGS outperformed all other options because it did not have any synchronization points, so it did not need to wait for slower processors. AGS and IGS on the asynchronous framework (IGS/A) slightly outperformed synchronous IGS (IGS/P) and the synchronous partitioned IGS (IGS/SP), thanks to lower communication overhead. As expected, IGS/A only scaled to 50 processors, because it could only perform 50 evaluations concurrently. IGS/S resulted in slower speeds on two and three clusters due to communication overhead from using the WAN. Since the different heterogeneous architectures execute Java at different speeds, the partitions used were sub-optimal, resulting in IGS/SP performing better as more clusters were added, but not by much. These

results show that using an asynchronous search can result in significant performance gains on heterogeneous environments.

## 6 Discussion

This paper describes an extension to the GMLE framework to support asynchronous master-work search methods on different large-scale heterogeneous environments. Two asynchronous master-worker versions of genetic search are introduced and evaluated. Results show that using asynchronous searching methods can result in significant performance gains on a heterogeneous grid environment and improved convergence rates. Additionally, the convergence rate and evaluation time using the asynchronous search for the MilkyWay@HOME project is comparable to a 1024 processor rack of a BlueGene supercomputer. These preliminary results do show that asynchronous master-worker genetic search is a promising method for heterogeneous architectures, and is an ideal candidate for the use over the BOINC Internet computing infrastructure, thanks to its fault tolerance, scalability, and lack of inter-worker communication.

The analysis of the different reproduction operators used on BOINC shows interesting results in that certain operators and slower processors tend to be more explorative in nature, while faster processors and other operators are more exploitative. An adaptive search could first focus on exploration by emphasizing certain operators and then focus on local search. It may also improve convergence rates to assign members generated by explorative operators to slower computers, and exploitative operators to faster ones.

Currently, the MilkyWay@HOME BOINC project consists of over 1,000 volunteered computers, providing a very large and heterogeneous network to study asynchronous search methods, as well as a powerful computing environment to use in modeling the MilkyWay galaxy. We hope to increase membership to test the limits of the scalability of these approaches. This work evaluated single population, or panmictic versions of asynchronous genetic search. Asynchronous island

(multi-population) versions will be of interest, especially if multiple servers are required to handle the load from a large BOINC community.

It was also shown that using a different type of reproduction strategy (double shot) resulted in a greatly improved convergence rate. Evaluating different reproduction and mutation strategies for asynchronous search could result in even better performing algorithms. Lastly, the algorithms presented were only tested on a single scientific application. Evaluating the algorithms with different applications will provide a better understanding of how they converge to a solution.

## 7 Acknowledgements

We would like to thank our many volunteers for taking part in the MilkyWay@HOME BOINC computing project as this research would not be possible without them.

This work has been partially supported by the following grants: NSF AST No. 0607618, NSF IIS No. 0612213, NSF MRI No. 0420703 and NSF CAREER CNS Award No. 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.

## 8 Biographies

**Travis Desell** Travis Desell is a Ph.D. student in the computer science department at RPI. He received his B.S. and M.S. degree from RPI in 2003 and 2007. His research interests include distributed computing, machine learning and asynchronous programming languages and middleware for large scale computing.

**Boleslaw Szymanski** Dr. Boleslaw Szymanski is the Claire and Roland Schmitt Distinguished Professor of Computer Science and the Founding Director of the Center for Pervasive Computing and Networking at RPI. He received his Ph.D. in Computer Science from National Academy of Sciences in Warsaw, Poland, in 1976 and M.Eng. in Electronics from Warsaw Polytechnic in 1973. He

is an author and co-author of over three hundred publications and an editor of three books. Dr. Szymanski is also an Editor-in-Chief of Scientific Programming, an IEEE Fellow and the member of the ACM for which he was National Lecturer. Dr. Szymanski interests focus on parallel and distributed computing and networking.

**Carlos A. Varela** Dr. Carlos A. Varela is an Associate Professor at the Department of Computer Science and Founding Director of the Worldwide Computing Laboratory, at RPI. He received his Ph.D., M.S., and B.S. degrees in Computer Science at UIUC in 2001, 2000, and 1992. Dr. Varela is Associate Editor and Information Director of the ACM Computing Surveys journal and has served as Guest Editor of the Scientific Programming journal. Dr. Varela received the National Science Foundation CAREER award in 2005. His current research interests include adaptive middleware and programming technology for distributed computing over wide-area networks.

## References

- [1] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.
- [2] E. Alba and B. Dorronsoro. The exploration/exploitation tradeoff in dynamic cellular genetic algorithms. *IEEE Transactions on Evolutionary Computation*, 9:126–142, April 2005.
- [3] E. Alba and J. M. Troya. Analyzing synchronous and asynchronous parallel distributed genetic algorithms. *Future Generation Computer Systems*, 17:451–465, January 2001.
- [4] D. P. Anderson, E. Korpela, and R. Walton. High-performance task distribution for volunteer computing. In *e-Science*, pages 196–203. IEEE Computer Society, 2005.
- [5] J. Berntsson and M. Tang. A convergence model for asynchronous parallel genetic algorithms. In *IEEE Congress on Evolutionary Computation (CEC2003)*, volume 4, pages 2627–2634, December 2003.
- [6] 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.
- [7] R. Buyya, D. Abramson, and J. 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.
- [8] E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10(2):141–171, 1998.
- [9] T. Desell, N. Cole, M. Magdon-Ismael, H. Newberg, B. Szymanski, and C. Varela. Distributed and generic maximum likelihood evaluation. In *3rd IEEE International Conference on e-Science and Grid Computing (eScience2007)*, page 8pp, Bangalore, India, December 2007. to appear.
- [10] B. Dorronsoro and E. Alba. A simple cellular genetic algorithm for continuous optimization. *IEEE Congress on Evolutionary Computation (CEC2006)*, pages 2838–2844, July 2006.
- [11] B. Dorronsoro, E. Alba, M. Giacobini, and M. 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.
- [12] G. Folino, A. Forestiero, and G. Spezzano. A JXTA based asynchronous peer-to-peer implementation of genetic programming. *Journal of Software*, 1:12–23, August 2006.
- [13] 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.
- [14] L. Gong. Jxta: A network programming environment. *IEEE Internet Computing*, 5:88–95, May/June 2001.
- [15] H. Imade, R. Morishita, I. Ono, N. Ono, and M. Okamoto. A grid-oriented genetic algorithm framework for bioinformatics. *New Generation Computing: Grid Systems for Life Sciences*, 22:177–186, January 2004.
- [16] A. Lewis and D. Abramson. An evolutionary programming algorithm for multi-objective optimization. In *IEEE Congress on Evolutionary Computation (CEC2003)*, volume 3, pages 1926–1932, December 2003.
- [17] D. Lim, Y.-S. Ong, Y. Jin, B. Sendhoff, and B.-S. Lee. Efficient hierarchical parallel genetic algorithms using grid computing. *Future Generation Computer Systems*, 23:658–670, May 2007.
- [18] T. Peachey, D. Abramson, and A. Lewis. Model optimization and parameter estimation with Nimrod/O. In *International Conference on Computa-*

*tional Science*, University of Reading, UK, May 2006.

- [19] J. Purnell, M. Magdon-Ismail, and H. J. Newberg. A probabilistic approach to finding geometric objects in spatial datasets of the Milky Way. In *Foundations of Intelligent Systems*, volume 3488/2005, pages 485–493. Springer Berlin / Heidelberg, 2005.
- [20] A. Sinha and D. E. Goldberg. A survey of hybrid genetic and evolutionary algorithms. Technical Report No. 2003004, Illinois Genetic Algorithms Laboratory (IlliGAL), 2003.