**Scaling Genetic Algorithms using MapReduce**

**Abhishek Verma, Xavier Llorà,**
**Roy H. Campbell, and David E. Goldberg**
**IlliGAL Report No. 2009007**
**October, 2009**

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue
Urbana, IL 61801
Office: (217) 333-2346
Fax: (217) 244-5705

# Scaling Genetic Algorithms using MapReduce

**Abhishek Verma[†], Xavier Llorà[‡], Roy H. Campbell[†], David E. Goldberg[⋆]**

[†]Department of Computer Science
University of Illinois at Urbana-Champaign
201 North Goodwin Avenue, Urbana, IL 61801
{verma7,rhc}@illinois.edu

[‡]National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street, Urbana, IL 61801
xllora@illinois.edu

[⋆]Industrial and Enterprise Systems Engineering
University of Illinois at Urbana-Champaign
104 S. Mathews, Urbana, IL 6180
deg@illinois.edu



October 9, 2009

**Abstract**

Genetic algorithms(GAs) are increasingly being applied to large scale problems. The traditional MPI-based parallel GAs do not scale very well. MapReduce is a powerful abstraction developed by Google for making scalable and fault tolerant applications. In this paper, we mould genetic algorithms into the the MapReduce model. We describe the algorithm design and implementation of GAs on Hadoop, the open source implementation of MapReduce. Our experiments demonstrate the convergence and scalability upto $10^5$ variable problems. Adding more resources would enable us to solve even larger problems without any changes in the algorithms and implementation.

## 1 Introduction

The growth of the internet has pushed researchers from all disciplines to deal with volumes of information where the only viable path is to utilize data-intensive frameworks (Uysal et al., 1998; Beynon et al., 2000; Foster, 2003; Mattmann et al., 2006). Genetic Algorithms are increasingly being used for large scale problems like non-linear optimization (Gallagher and Sambridge, 1994), clustering (Frnti et al., 1997) and job scheduling (Sannomiya et al., 1999). The inherent parallel nature of evolutionary algorithms makes them optimal candidates for parallelization (Cantú-Paz, 2000). Although large bodies of research on parallelizing evolutionary computation algorithms are available (Cantú-Paz, 2000), there has been little work done in exploring the usage of data-intensive computing.

The main contributions of the paper are as follows:

- We demonstrate a transformation of Genetic Algorithms into the Map and Reduce primitives
- We implement the MapReduce program and demonstrate its scalability to large problem sizes.

The organization of the paper is as follows: We introduce the MapReduce model and its execution overview in Section 2. Then, we discuss how genetic algorithms can be moulded into the MapReduce

model in Section 3 and report our experiments in Section 4. In Section 5, we discuss the related work and finally conclude with Section 6.

# 2 MapReduce

Inspired by the *map* and *reduce* primitives present in functional languages, Google developed the MapReduce (Dean and Ghemawat, 2008) abstraction that enables the users to easily develop large-scale distributed applications. The associated implementation parallelizes large computations easily as each map function invocation is independent and uses re-execution as the primary mechanism of fault tolerance.

In this model, the computation takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce. Map, written by the user, takes an input pair and produces a set of intermediate key/value pairs. The MapReduce framework then groups together all intermediate values associated with the same intermediate key $I$ and passes them to the Reduce function. The Reduce function, also written by the user, accepts an intermediate key $I$ and a set of values for that key. It merges together these values to form a possibly smaller set of values. The intermediate values are supplied to the user's reduce function via an iterator. This allows the model to handle lists of values that are too large to fit in main memory.

Conceptually, the map and reduce functions supplied by the user have the following types:

$$\begin{aligned} map(k_1, v_1) &\rightarrow list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow list(v_3) \end{aligned}$$

i.e., the input keys and values are drawn from a different domain than the output keys and values. Furthermore, the intermediate keys and values are from the same domain as the output keys and values.

The Map invocations are distributed across multiple machines by automatically partitioning the input data into a set of $M$ splits. The input splits can be processed in parallel by different machines. Reduce invocations are distributed by partitioning the intermediate key space into $R$ pieces using a partitioning function, which is $hash(key)\%R$ according to the default Hadoop configuration (which we later override for our needs). The number of partitions ($R$) and the partitioning function are specified by the user. Figure 1 shows the high level data flow of a MapReduce operation. Interested readers may refer to Dean and Ghemawat (2008) for the details.

# 3 MapReducing GAs

In this section, we start with a simple model of Genetic algorithms and then transform and implement it using MapReduce along with a discussion of some of the elements that need to be taken into account. We encapsulate each iteration of the GA as a seperate MapReduce job. The client accepts the commandline parameters, creates the population and submits the MapReduce job.

## 3.1 Genetic Algorithms

Selectorecombinative genetic algorithms (Goldberg, 1989, 2002), one of the simplest forms of GAs, mainly rely on the use of selection and recombination. We chose to start with them because they present a minimal set of operators that help us illustrate the creation of a data-intensive flow

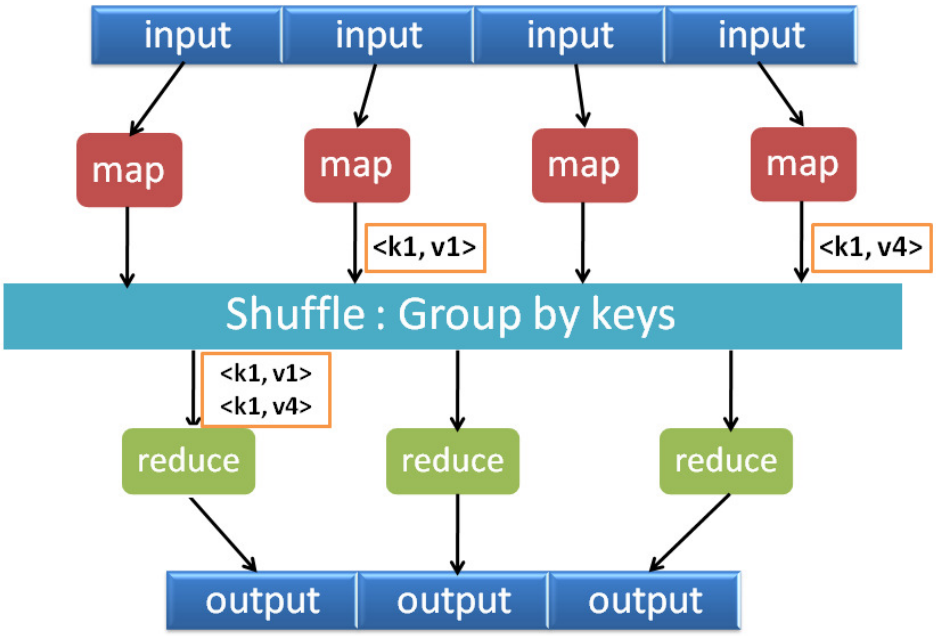


Figure 1: MapReduce Data flow overview

counterpart. The basic algorithm that we target to implement as a data-intensive flow can be summarized as follows:

1. Initialize the population with random individuals.
2. Evaluate the fitness value of the individuals.
3. Select good solutions by using s-wise tournament selection without replacement (Goldberg et al., 1989b).
4. Create new individuals by recombining the selected population using uniform crossover[1] (Syswerda, 1989).
5. Evaluate the fitness value of all offspring.
6. Repeat steps 3–5 until some convergence criteria are met.

### 3.2 Map

Evaluation of the fitness function for the population (Steps 1 and 5) matches the MAP function, which has to be computed independent of other instances. As shown in the algorithm in Listing 1, the MAP evaluates the fitness of the given individual. Also, it keeps track of the the best individual and finally, writes it to a global file in the Distributed File System (HDFS). The client, which has initiated the job, reads these values from all the mappers at the end of the MapReduce and checks if the convergence criteria has been satisfied.

[1]We assume a crossover probability $p_\chi$=1.0.

Listing 1: Map phase of each iteration of the Genetic Algorithm

```
procedure Initialization:
begin
   max := −1
end

procedure Map(key, value):
begin
   individual :=
        Individual_representation(key)
   fitness := CalculateFitness(individual)
   Emit (individual, fitness)
 {Keep track of the current best}
   if fitness>max then
      max := fitness
      maxInd := individual
 {Finished all local maps}
   if processed_all_individuals then
      Write best individual to global file in DFS
end
```

### 3.3 Partitioner

If the selection operation in a GA (Step 3) is performed locally on each node, it reduces the selection pressure (Sarma et al., 1998) and can lead to increase in the time taken to converge. Hence, decentralized and distributed selection algorithms (Jong and Sarma, 1995) are preferred. The only point at which there is a global communication is in the shuffle between the Map and Reduce. At the end of the Map phase, the MapReduce framework shuffles the key/value pairs to the reducers using the partitioner. The partitioner splits the intermediate key/value pairs among the reducers. The function GETPARTITION() returns the reducer to which the given $(key, value)$ should be sent to. In the default implementation, it uses HASH($key$) % $numReducers$ so that all the values corresponding to a given $key$ end up at the same reducer which can then apply the REDUCE function. However, this does not suit the needs of Genetic algorithms because of two reasons: Firstly, the HASH function partitions the namespace of the individuals $N$ into $r$ distinct classes : $\{N_0, N_1, \ldots, N_{r-1}\}$ where $N_i = \{n : \text{HASH}(n) = i\}$. The individuals within each partition are isolated from all other partitions. Thus, the HASHPARTITIONER introduces an artificial spatial constraint based on the lower order bits. Because of this, the convergence of the genetic algorithm may take more iterations or it may never converge at all.

Secondly, as the genetic algorithm progresses, the same (close to optimal) individual begins to dominate the population. All copies of this individual will be sent to one single reducers which will get overloaded. Thus, the distribution progressively becomes more skewed, deviating from the uniform distribution (that would have maximized the usage of parallel processing). Finally, when the GA converges, all the individuals will be processed by that single reducer. Thus, the parallelism decreases as the GA converges and hence, it will take more iterations.

For these reasons, we override the default partitioner by providing our own partitioner, which shuffles individuals randomly across the different reducers as shown in Listing 2.

Listing 2: Random partitioner for the Genetic Algorithm

```
int getPartition(key, value, numReducers):
    return RandomInt(0, numReducers - 1)
```

### 3.4 Reduce

We implement Tournament selection without replacement (Goldberg et al., 1989a). A tournament is conducted among $tSize$ randomly chosen individuals and the winner is selected. This process is repeated *population* number of times. Since randomly selecting individuals is equivalent to randomly shuffling all individuals and then processing them sequentially, our reduce function goes through the individuals sequentially. Initially the individuals are buffered for the last rounds, and when the tournament window is full, SelectionAndCrossover is carried out as shown in the Listing 3. When the crossover window is full, we use the Uniform Crossover operator. For our implementation, we set the $tSize$ to 5 and the $cSize$ to 2.

### 3.5 Optimizations

After initial experimentation, we noticed that for larger problem sizes, the serial initialization of the population takes a long time. According to Amdahl's law, the speedup is bounded because of this serial component. Hence, we create the initial population in a separate MapReduce phase, in which the Map generates random individuals and the Reduce is the Identity Reducer. [2] We seed the pseudo-random number generator for each mapper with $mapperId \cdot currentTime$. The bits of the variables in the individual are compactly represented in an array of **long long int**s and we use efficient bit operations for crossover and fitness calculations. Due to the inability of expressing loops in the MapReduce model, each iteration consisting of a Map and Reduce, has to executed till the convergence criteria is satisfied.

## 4 Results

We implemented the simple OneMAX problem on Hadoop (0.19)[3] and ran it on our 416 core (52 nodes) Hadoop cluster. Each node runs a two dual Intel Quad cores, 16GB RAM and 2TB hard disks. The nodes are integrated into a Distributed File System (HDFS) yielding a potential single image storage space of $2 \cdot 52/3 = 34.6TB$ (since the replication factor of HDFS is set to 3). A detailed description can be found elsewhere[4]. Each node can run 5 mappers and 3 reducers in parallel. Some of the nodes, despite being fully functional, may be slowed down due to disk contention, network traffic, or extreme computation loads. Speculative execution is used to run the jobs assigned to these slow nodes, on idle nodes in parallel. Whichever node finished first, writes the output and the other speculated jobs are killed. For each experiment, the population for the GA is set to $n \log n$ where n is the number of variables.

We perform the following experiments:

1. **Convergence Analysis:** In this experiment, we monitor the progress in terms of the number of bits set to 1 by the GA for a $10^4$ variable OneMAX problem. As shown in Figure 2, the GA converges in 220 iterations taking an average of 149 seconds per iteration.

[2] Setting the number of reducers to 0 in Hadoop removes the extra overhead of shuffling and identity reduction.
[3] http://hadoop.apache.org
[4] http://cloud.cs.illinois.edu

Listing 3: Reduce phase of each iteration of the Genetic Algorithm

```
procedure Initialization:
begin
    processed := 0
    Allocate tournamentArray [1 ... 2*tSize]
    Allocate crossoverArray [cSize]
end

procedure Reduce(key, values):
begin
    while values.hasNext()
    begin
        individual := Individual_representation(key)
        fitness := values.getValue()
        if processed<tSize
        then
          {Wait for individuals to join in the tournament and put them for the
              last rounds}
           tournamentArray [tSize + processed%tSize] := individual
        else
          {Conduct a tournament over the past window}
           SelectionAndCrossover()
        processed := processed + 1

          {Finished all reduces}
        if processed_all_individuals
        then
          {Cleanup for the last tournament windows}
           for k=1 to tSize
           begin
              SelectionAndCrossover()
              processed := processed + 1
           end
    end
end

procedure SelectionAndCrossover:
begin
    crossoverArray [processed%cSize] := Tournament(tournamentArray)
    if (processed-tSize)%cSize = cSize - 1
    then
          {Perform crossover whenever the crossover window is full}
        newIndividuals := Crossover(crossoverArray)
        for individual in newIndividuals
        Emit(individual, dummyFitness)
end
```

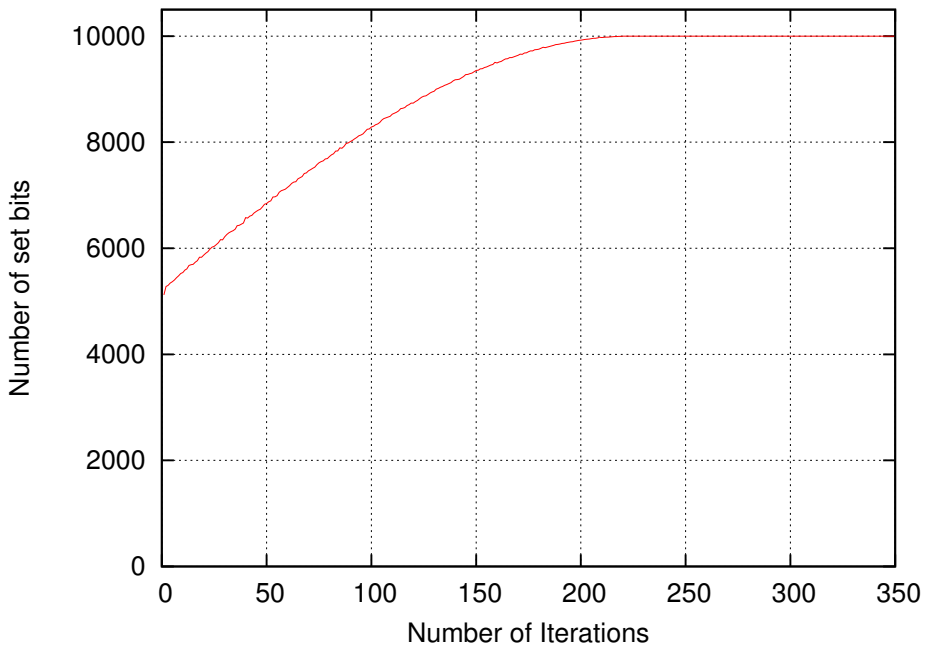


Figure 2: Convergence of Genetic algorithm for the $10^4$ variable OneMAX problem

2. **Scalability with constant load per node:** In this experiment, we keep the load set to 1,000 variables per mapper. As shown in Figure 3, the time per iteration increases initially and then stabilizes around 75 seconds. Thus, increasing the problem size as more resources are added does not change the iteration time. Since, each node can run a maximum of 5 mappers, the overall map capacity is $5 \cdot 52(nodes) = 260$. Hence, around 250 mappers, the time per iteration increases due to the lack of resources to accommodate so many mappers.

3. **Scalability with constant overall load:** In this experiment, we keep the problem size fixed to 50,000 variables and increase the number of mappers. As shown in Figure 4, the time per iteration decreases as more and more mappers are added. Thus, adding more resources keeping the problem size fixed decreases the time per iteration. Again, saturation of the map capacity causes a slight increase in the time per iteration after 250 mappers. However, the overall speedup gets bounded by Amdahl's law introduced by Hadoop's overhead (around 10s of seconds to initiate and terminate a MapReduce job). However, as seen in the previous experiment, the MapReduce model is extremely useful to process large problems size, where extremely large populations are required.

4. **Scalability with increasing the problem size:** Here, we utilize the maximum resources and increase the number of variables. As shown in Figure 5, our implementation scales to $n = 10^5$ variables, keeping the population set to $n \log n$. Adding more nodes would enable us to scale to larger problem sizes. The time per iteration increases sharply as the number of variables is increased to $n = 10^5$ as the population increases super-linearly ($n \log n$), which is more than 16 million individuals.

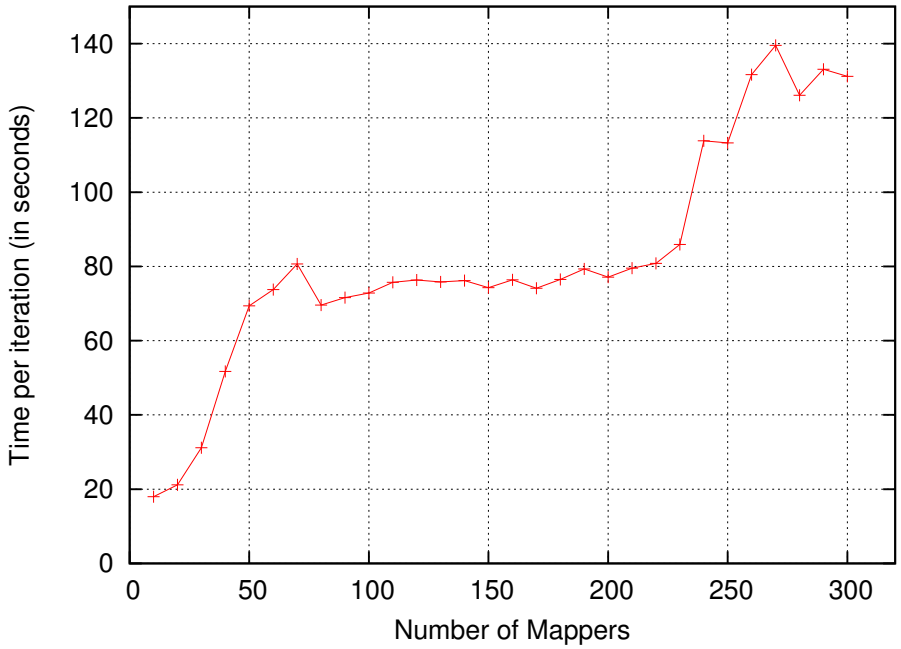


Figure 3: Scalability of Genetic algorithm with constant load per node for the OneMAX problem

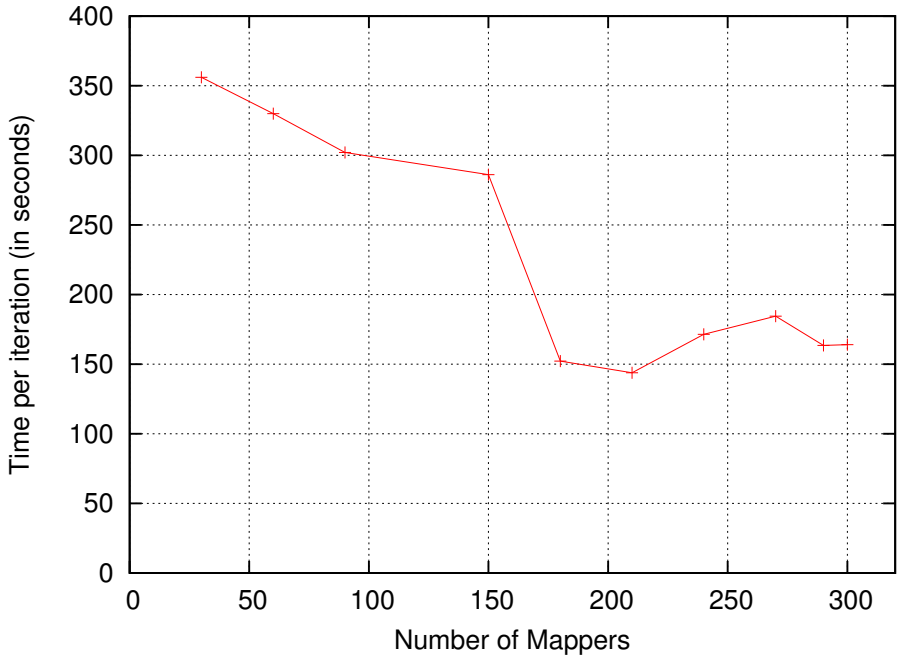


Figure 4: Scalability of Genetic algorithm for the 50,000 variable OneMAX problem with increasing number of mappers

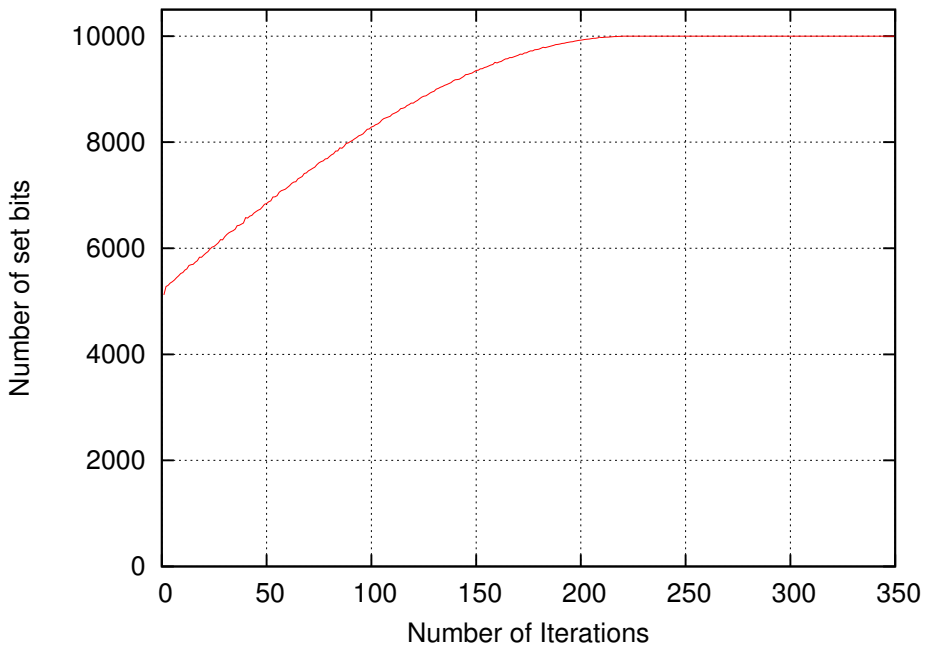


Figure 5: Scalability of Genetic algorithm for OneMAX problem with increasing number of variables

## 5 Related Work

Several different models like fine grained (Maruyama et al., 1993), coarse grained (Lin et al., 1994) and distributed models (Lim et al., 2007) have been proposed for implementing parallel GAs. Traditionally, MPI has been used for implementing parallel GAs. However, MPIs do not scale well on commodity clusters where failure is the norm, not the exception. Generally, if a node in an MPI cluster fails, the whole program is restarted. In a large cluster, a machine is likely to fail during the execution of a long running program, and hence fault tolerance is necessary.

MapReduce (Dean and Ghemawat, 2008) is a programming model that enables the users to easily develop large-scale distributed applications. Hadoop is an open source implementation of the MapReduce model. Several different implementations of MapReduce have been developed for other architectures like Phoenix (Raghuraman et al., 2007) for multicores and CGL-MapReduce (Ekanayake et al., 2008) for streaming applications.

To the best of our knowledge, MRPGA (Jin et al., 2008) is the only attempt at combining MapReduce and GAs. However, they claim that GAs cannot be directly expressed by MapReduce, extend the model to MapReduceReduce and offer their own implementation. We point out several shortcomings: Firstly, the Map function performs the fitness evaluation and the "ReduceReduce" does the local and global selection. However, the bulk of the work - mutation, crossover, evaluation of the convergence criteria and scheduling is carried out by a single co-ordinator. As shown by their results, this approach does not scale above 32 nodes due to the inherent serial component. Secondly, the "extension" that they propose can readily be implemented within the traditional MapReduce model. The local reduce is equivalent to and can be implemented within a Combiner (Dean and Ghemawat, 2008). Finally, in their **mapper**, **reducer** and **final_reducer** functions, they emit "$default\_key$" and 1 as their values. Thus, they do not use any characteristic of the

MapReduce model - the grouping by keys or the shuffling. The Mappers and Reducers might as well be independently executing processes only communicating with the co-ordinator.

We take a different approach, trying to hammer the GAs to fit into the MapReduce model, rather than change the MapReduce model itself. We implement GAs in Hadoop, which is increasingly becoming the de-facto standard MapReduce implementation and used in several production environments in the industry. Meandre (Llorà et al., 2008) extends beyond some limitations of the MapReduce model while maintaining a data-intensive nature. It shows linear scalability of simple GAs and EDAs on multicore architectures. For very large problems ($> 10^9$ variables), other models like Compact Genetic Algorithms(cGA) and Extended cGA(eCGA) have been explored (Sastry et al., 2007).

## 6 Conclusions and Future Work

In this paper, we have mainly addressed the challenge of using the MapReduce model to scale genetic algorithms. We described the algorithm design and implementation of GAs on Hadoop. The convergence and scalability of the implementation has been investigated. Adding more resources would enable us to solve even larger problems without any changes in the algorithm implementation.

General Purpose GPUs are an exciting addition to the heterogenity of clusters. The compute intensive Map part and the random number generation can be scheduled on the GPUs, which can be performed in parallel with the Reduce on the CPUs. MapReducing more scalable GA models like Compact GAs and Extended Compact GAs will be investigated in future.

## Acknowledgments

This work was funded, in part, by NSF IIS Grant #0841765. We would also like to thank the National Center for Supercomputing Applications for the committed support that has made possible this collaboration.

## References

Beynon, M. D., Kurc, T., Sussman, A., and Saltz, J. (2000). Design of a framework for data-intensive wide-area applications. In *HCW '00: Proceedings of the 9th Heterogeneous Computing Workshop*, page 116, Washington, DC, USA. IEEE Computer Society.

Cantú-Paz, E. (2000). *Efficient and Accurate Parallel Genetic Algorithms*. Springer.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Ekanayake, J., Pallickara, S., and Fox, G. (2008). Mapreduce for data intensive scientific analyses. In *ESCIENCE '08: Proceedings of the 2008 Fourth IEEE International Conference on eScience*, pages 277–284, Washington, DC, USA. IEEE Computer Society.

Foster, I. (2003). The virtual data grid: A new model and architecture for data-intensive collaboration. In *in the 15$^{th}$ International Conference on Scientific and Statistical Database Management*, pages 11–.

Frnti, P., Kivijrvi, J., Kaukoranta, T., and Nevalainen, O. (1997). Genetic algorithms for large scale clustering problems. *Comput. J*, 40:547–554.

Gallagher, K. and Sambridge, M. (1994). Genetic algorithms: a powerful tool for large-scale nonlinear optimization problems. *Comput. Geosci.*, 20(7-8):1229–1236.

Goldberg, D., Deb, K., and Korb, B. (1989a). Messy genetic algorithms: motivation, analysis, and first results. *Complex Systems*, (3):493–530.

Goldberg, D. E. (1989). *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA.

Goldberg, D. E. (2002). *The Design of Innovation: Lessons from and for Competent Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA.

Goldberg, D. E., Korb, B., and Deb, K. (1989b). Messy genetic algorithms: Motivation, analysis, and first results. *Complex Systems*, 3(5):493–530.

Jin, C., Vecchiola, C., and Buyya, R. (2008). Mrpga: An extension of mapreduce for parallelizing genetic algorithms. *eScience, 2008. eScience '08. IEEE Fourth International Conference on*, pages 214–221.

Jong, K. D. and Sarma, J. (1995). On decentralizing selection algorithms. In *In Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 17–23. Morgan Kaufmann.

Lim, D., Ong, Y.-S., Jin, Y., Sendhoff, B., and Lee, B.-S. (2007). Efficient hierarchical parallel genetic algorithms using grid computing. *Future Gener. Comput. Syst.*, 23(4):658–670.

Lin, S.-C., Punch, W. F., and Goodman, E. D. (1994). Coarse-grain parallel genetic algorithms: Categorization and new approach. In *Proceeedings of the Sixth IEEE Symposium on Parallel and Distributed Processing*, pages 28–37.

Llorà, X., Ács, B., Auvil, L., Capitanu, B., Welge, M., and Goldberg, D. E. (2008). Meandre: Semantic-driven data-intensive flows in the clouds. In *Proceedings of the 4th IEEE International Conference on e-Science*, pages 238–245. IEEE press.

Maruyama, T., Hirose, T., and Konagaya, A. (1993). A fine-grained parallel genetic algorithm for distributed parallel systems. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 184–190, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Mattmann, C. A., Crichton, D. J., Medvidovic, N., and Hughes, S. (2006). A software architecture-based framework for highly distributed and data intensive scientific applications. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 721–730, New York, NY, USA. ACM.

Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C. (2007). Evaluating mapreduce for multi-core and multiprocessor systems. *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*.

Sannomiya, N., Iima, H., Ashizawa, K., and Kobayashi, Y. (1999). Application of genetic algorithm to a large-scale scheduling problem for a metal mold assembly process. *Proceedings of the 38th IEEE Conference on Decision and Control*, 3:2288–2293.

Sarma, A. J., Sarma, J., and Jong, K. D. (1998). Selection pressure and performance in spatially distributed evolutionary. In *In Proceedings of the World Congress on Computatinal Intelligence*, pages 553–557. IEEE Press.

Sastry, K., Goldberg, D. E., and Llora, X. (2007). Towards billion-bit optimization via a parallel estimation of distribution algorithm. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 577–584, New York, NY, USA. ACM.

Sywerda, G. (1989). Uniform crossover in genetic algorithms. In *Proceedings of the third international conference on Genetic algorithms*, pages 2–9, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Uysal, M., Kurc, T. M., Sussman, A., and Saltz, J. (1998). A performance prediction framework for data intensive applications on large scale parallel machines. In *In Proceedings of the Fourth Workshop on Languages, Compilers and Run-time Systems for Scalable Computers, number 1511 in Lecture Notes in Computer Science*, pages 243–258. Springer-Verlag.