**Genetic Algorithms with Mapreduce Runtimes**

Fei Teng1, Doga Tuncay2

Indiana University Bloomington School of Informatics and Computing Department

CS PhD Candidate1, Masters of CS Student2

{feiteng,dtuncay}@indiana.edu

**Abstract**

Data-intensive Computing has played a key role in processing vast volumes of data exploiting massive parallelism. Parallel computing frameworks have proven that terabytes of data can be routinely processed. Mapreduce is a parallel programming model and associated implementation founded by Google, which is one of the leading companies in IT. Genetic Algorithms are increasingly being applied on parallel computing to large scale problems. Since GAs have the parallelism in their nature, they can be easily applied on Parallel Runtimes such as MPI, Hadoop Runtime and Twister[9]. Researches have shown that, Genetic Algorithms were a great success and might be modeled on MPI (Message Passing Interface) and Mapreduce Models. In this paper, the nature of Genetic Algorithms, how they can efficiently be used\* on Parallel computing networks and how they can be applied to Mapreduce Model is served. In the project, we will be appying Genetic Algorithm and we primarily use the basic algorithm Onemax Problem on Hadoop Mapreduce Framework and the Twister Iterative Mapreduce Framework. Possible to anticipate that the Twister Runtime could perform better, GAs would also perform better performance on Iterative Model from its nature and Twister’s computing model. On the grounds of our observations, Twister performs almost 25 times faster than Hadoop. The content of the paper includes a brief introduction to Mapreduce algorithm, an explanation to the simple Genetic Algorithm OneMax problem in the third section\* and in the fourth section it includes the architecture design of sequential and Mapreduce Genetic Algorithms. In the fifth section, we will give an in depth information of GA on Hadoop and Twister. Thereafter, performance analysis demonstration shall take place. In the following sections, we will be talking about a related work and concluding our project with discussions and a conclusion part.

**Keywords:** Genetic Algorithms, Onemax Problem, Mapreduce, Hadoop, Twister, Parallel Computing, Data-intensive Computing

1. **Introduction**

Genetic Algorithms are increasingly being used for solving large scaled problems, such as clustering[1], non-linear optimization [2] and job scheduling [3]. The parallel nature of Genetic Algorithms makes them an optimal base for parallelization. We will call Parallel Genetic Algorithms as PGAs and PGAs are not the only parallel version of serial GAs. In fact, they support an ideal base to have a parallel algorithm that behaves better than the sum of the separate behaviors of its component sub-algorithms[4].

An observation has shown us that the PGAs’ structured populations, either in the form of a set of islands or a diffusion grid leads to superior numerical performance which is a benefit for the system. Therefore, many parallel computing scientists do not use a parallel machine to run structure-population because of its parallel model and they still are able to get better results than serial GAs. [5] There is another way to parallelize the system which is the hardware parallelization, a way of speeding up the execution of the algorithm in hardware manners. Hence, a structured population model can be defined and implemented on any parallel machine. There are some researches on this topic that are showing a ring of panmictic GAs on MIMD/SIMD computers.

Beyond parallel machines, we are able to implement GA’s on Mapreduce Runtimes to improve the parallel behavior of the GAs. A research has shown that the implementation of simple and compact GAs of Hadoop demonstrate the convergence and scalability up to 105 to 108 variable problems respectively [6]. Therefore, the main contributions of this paper are: demonstrating that genetic algorithms can be transformed into the map and reduce primitives and can be implemented to a Mapreduce program with scalable results for the large problem size. We also support the idea that Twister Iterative Mapreduce Runtime will have a better performance, since its nature supports the design of evolutionary computing algorithms. Therefore, we will implement both Hadoop and Twister Runtime, by using the basic Genetic Algorithm Onemax and analyze the performance with a comparison between structures and designs of the Mapreduce algorithms with an environment-independent approach.

1. **Mapreduce**

Google introduced Mapreduce, inspired by the map and reduce primitives in functional languages. It is used to enable users to develop large-scale distributed applications. Mapreduce parallelizes large computations easily since each map function runs independently and provides fault tolerance through re-execution.

In Mapreduce, the input is a set of key/value pairs, and the output is a set key/value pairs. Mapreduce operation breaks down to two functions; map and reduce. Map takes an input pair and produces a set of intermediate key/value pairs. Then all intermediate values are grouped together by their intermediate keys. This is then passed onto the Reduce function. Reduce function takes an intermediate key and its related set of values, and merges them together to try to form a smaller set. An iterator supplies the list of values to the reduce function.

A partitioner automatically splits the data and distributes them to the map functions running across multiple machines, so the process is parallelized. Another partitioner handles splitting the intermediate key space and supplying them to the distributed reduce functions. Partitioner is configured by the user.

1. **Onemax Problem Statement**

Onemax problem is a basic optimization problem that aims to maximize the number of ones in a bit string. The bit strings are used to show genes with fixed length strings.

In Onemax problem, we formally describe this problem as finding a string

and *xi* ∈ {0,1}

which maximizes the following sum equation,

where x is a gene bit string with a fixed length n = |x|. We consider that there is no noise involved with our gene encoding.

Onemax represents an optimization problem with independent variables and it is widely used as benchmark to measure the effectiveness and efficiency of a class of genetic algorithms.

1. **Architecture design**

To solve the Onemax problem described in section 1, we need efficient computing architecture for Mapreduce genetic algorithm. However, before put genetic algorithm under the Mapreduce context, we must have a thorough understanding about the genetic algorithm for Onemax problem. For this sake, we will first give out the design of serial Onemax GA and then present how to program the same GA in Mapreduce context to tackle large-scale Onemax application.

***4.1 Serial Onemax genetic algorithm***

* + - *Gene representative*

It’s not hard to represent a gene for Onemax problem. Traditionally, a gene in Onemax population is represented as a bit string. For instance, if we set the problem size as 10, which means the maximum length of a gene is 10, then we can have a legal gene representative as a binary number with length of 10 like this <1101110010>.

* *Genetic operators*

We use tournament selection[7] without replacement as the select operator. The tournament size is set as 5 initially. We will run experiments to tune the parameter to make it optimal for our problem size[6] which shows that uniform crossover[8] is well applicable for Onemax problem with binary gene representative. So we choose uniform crossover as well. At last, to guarantee a fast algorithm converging, we don’t need a large mutation rate. Currently, we set mutation rate as 0.01.

* *Fitness function*

The fitness function is explicitly given in the problem statement. And the fitness value is how many one appears in the gene representative.

* *Serial GA flow chart*

Based on what’s mentioned above, we give out the GA flow chart as Figure 1.

For a population with size N, we conduct N select operator and use uniform crossover to derive N new better offspring in terms of fitness value. If we find that population in generation K is converged and meets the stop criterion, then we output the final optimal gene from the final population.

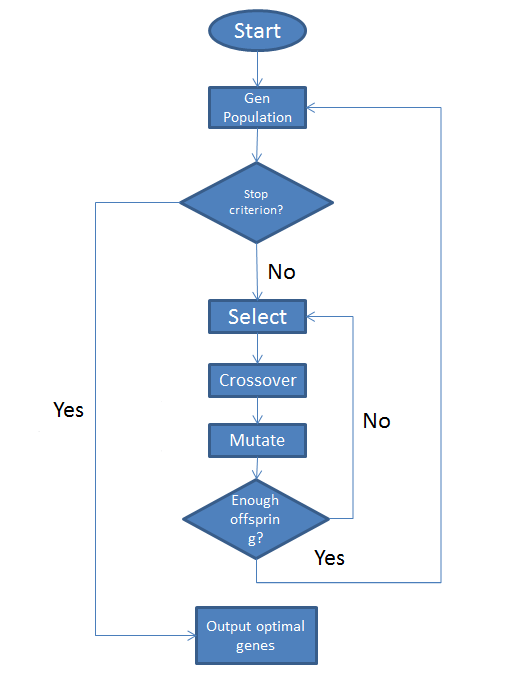


Figure 1. Serial GA flow chart

***4.2 Mapreduce Onemax genetic algorithm***

*4.2.1 Basic Mapreduce GA design in Twister*

Based on the serial GA, we can naturally come up with the Mapper and Reducer design. At first, we will create a new value type for the gene representative mentioned above. Then we can have the <key, value> pair in this form, <GeneUid, GeneValue>. What’s worthy of attention here is that we don’t simply choose the binary gene representative as the key, but to generate an Uid for each gene. The reason underlying this design is that with the evolution of the whole population, most of the genes will have the same representative because all genes have the pressure to converge on the single optimal solution for Onemax problem. Thus, if we trivially use binary representative as the key, almost all the genes in the population will be assigned to one single reducer (reducer receives its input from mappers based on the key). It’s really undesirable design with respect to load balance. The uid of a gene is randomly generated and if we choose a big enough random range, the probability of two genes having the same uid is very small. In this way, we can archive nearly even load balance.

The reducer will be in charge of conducting the select and crossover operation. After each reducer emits all new offspring to combiner, the combiner will partition the new population into M parts for M mappers to consume. In this way, the iterative nature of GA is embodied well in Mapreduce context. What’s more, twister explicitly supports iterative Mapreduce. So implementation complexity is reduced a lot with the twister iterative API.

*4.2.2 Optimization*

We aim to resolve large scale Onemax problem, so the size of population and the length of a gene representative will be much larger than what serial GA can handle. For example, we want to tackle a Onemax problem with a 105 size, then the corresponding population size will be 104 (the relation between problem size and population size please reference [7]). In this way, each individual gene will need 100KB memory space by using plain text input format. The whole population will occupy 100K\*10,000=1GB space. Then the communication cost will be the potential bottleneck.

The Onemax problem is very easy to validate because the optimal solution is obvious. If our Mapreduce GA can give out the size of the problem, e.g, the length of the gene representative, then our results can be declared as correct. Besides, we will also do a performance validation for Mapreduce GA with some parallelism metric like speed-up to prove that Mapreduce is not only correct but efficient. Some performance charts based on different problem scale will be given as well to demonstrate the scalability of Mapreduce GA.

1. **OneMax Implementation on Mapreduce Runtimes**
   1. ***Hadoop Genetic Algorithms***

Genetic Algorithm on Hadoop works smoothly and gives us very accurate results. We accomplished this by making Hadoop support iterative MapRaduce. Since Hadoop Mapreduce doesn’t use an iterative model, we needed to start a new job for each iteration to make Hadoop act an iterative behavior. Then we put the output in HDFS iteratively. We needed to override the interfaces to make customized value type. Hadoop Mapreduce doesn’t know the system of the GA by overriding the interfaces Hadoop understands the customized value types and so on.

Here is the data flow chart of the Hadoop:

**JobTracker**

**Reducers**

Initial population

Sub populations

**Mappers**

**…**

HDFS

HDFS

Figure 2. Hadoop GA Dataflow

*5.1.1 Hadoop Map*

In the mapper we get the initialized and partitioned data from HDFS and we calculate the Fitness Value and we return the IntWrite and the Gene for reduce use.

**-------------------------------------------------------------------------------------------------------------------**

**Algorithm 1** Hadoop Map Phase

*Map(filename, hdfsFilePath)*

**-------------------------------------------------------------------------------------------------------------------**

1: **for** i <- 0 **to** subPopSize

2: read <- geneBytes

3: bs <- geneLen

4**: for** j <- 0 **to** geneLen

5: **if** (geneBytes[j/SIZE] **and** 1<<(j%8)) > 0 **then**

6 **bs.set()**

7: **end if**

8: **end for**

9: gene <- new gene

10**:** gene.calcFitValue()

11**:** context.write <- IntWritable(i+1), gene

12**: end if**

*5.1.2 Hadoop Reduce*

We input the <IntWritable, Gene> key value pair from Mappers and we do the the evolving here to find the best offsprings in each subpopulation we return the outputs of the reducer back to the HDFS and after all subpopulations are mapped we return the output to reducers then we output the best offspring.

**-----------------------------------**

**Algorithm 2** Reduce Phase of each iteration of the GA

*Reduce(IntWritable, Gene)*

**-----------------------------------**

1: reduceNo <- key.get() % context.getNumReduceTasks()

2**: for** Gene **in** values

3: Gene <- new Gene

4:gene.setKey <- value.getKey())

5:gene.setFitValue <-value.getFitValue()

6:subPoP.addGene <- gene

7: **end for**

8**: Cleanup:**

9**:** subPop.setAvgFitvalue <- subPop.calcAvgFitValue

10**:** subPop.startEvolve()

11**:** *{Write the new population into the reduce output file}*

12**:** create dataOutputStream

13**:** outfile <- subPop.getAvgFitValue()

14**: for** i <- 0 **to** subPop.getPopSize()

15**:** gene <- subPop.getGenePool().get(i)

16**: for** j <- 1**to** gene.getKey()

17**: if** (geneBytes[j/SIZE] **and** 1<<(j%8)) > 0 **then**

18**: set** j

19**: end if**

20**: end for**

21**:** dataOutputStream <- bytesOfGene

22**: end for**

* 1. ***Twister Genetic Algorithms***

Twister supports the iterative semantic by its nature and the Genetic Algorithm has an iterative nature, which gives us a perfect match to proceed on. Twister doesn’t have a file system and hard disk I/O involved, uses static data, which makes a faster access to the files that is not the case in Hadoop. Also Twister uses Combiner to restore the next generation population. We again override interfaces to make new value type.

Here is the dataflow chart of Twister:

Sub population

**.**

**.**

**.**

Sub population

Sub population

**.**

**.**

**.**

**.**

**.**

**.**

Figure 3. Twister Dataflow Architecture

* + 1. *Twister Mapper*

**----------------------------------------------------------------------------------------------------------------**

**Algorithm 3** Twister Mapper Phase

*Map(collector, key, val)*

**----------------------------------------------------------------------------------------------------------------**

1: **subPop <- (Population)val**

2: Vector gene <- subPop.getGenePool()

3: Random rand

4**: for** i <- 0 **to** vecGene.size

5: gene <- vecGene.get(i)

6**:** gene.setGeneId(rand.nextInt())

7:gene.calcFitValue()

8: **end for**

9: keyNo <- 0

10**: for** gene : subPop.getGenePool()

11**:** collect <- (new StringKey(keyNo++ % this.numReduceTasks)), gene

12**: end for**

* + 1. *Twister Reducer*

**-----------------------------------**

**Algorithm 4** Twister Reducer Phase

*Reduce(collector, key, values)*

**-----------------------------------**

1: **if** values.size <= 0 **then**

2: throw exception

3: **end if**

4**:**  popSize = values.Size()

5:  **for** value : values

6**: subPop.addGene((Gene)value)**

7: **end for**

8: subPop.startEvolve()

9: collect <- (new StringKey(subPop.getCurrentGeneration())), subPop

Other than Hadoop, after Map and Reduce Phases are completed we need a combiner to combine the results with the (Map<Key, Value> keyValues) pairs that are gotten from the reducers and produces the final offsprings.

1. **Performance Evaluation**

We test our simple genetic algorithm on two different clusters because computing resources are limited especially during the last few weeks. However, it does not affect the accuracy of our performance evaluation because the evaluation is based on comparison and scalability. The performance testing is divided into two parts, the first is the performance comparison between Twister GA and Hadoop GA; the second is the scalability of Twister GA.

***6.1 Performance comparison between Hadoop and Twister***

Testing environment – Futuregrid

* 8 nodes x 8 cores
* CPU: 2.93GB for each core
* Memory: 24GB

The gene length used in the testing is 16384 bits (2KB); the population size is 5120 and converging criterion is to find the gene with the max one number, that’s 16384. The testing result is showed in Figure 4.

Figure 4. Hadoop/Twister GA execution time comparison

As we expected before, Twister GA is much faster than Hadoop GA. The reason is obvious, in the application, Hadoop must store the output from each generation to HDFS and then retrieve the output from last generation for the current generation as the input. All these HDFS operations involve much Hard disk I/O, which is very time consuming compared to memory and cache I/O. However, Hadoop has no way to go expect doing HDFS to support iterative Mapreduce. In contrast, because the build-in iterative semantics of Twister, iterative application like GAs can be implemented in a natural way without the file system support once developers have a clear understanding about what are static data and the data flow of dynamic data. Twister caches all static data in memory and the dynamic data, the genes and populations in simple GA, is stored in main memory of each nodes and transferred using optimized communication algorithm based on TCP protocol, which is also much faster compared to the HTTP transfer method of Hadoop. All of these contribute to the impressive performance comparison results displayed in Figure 4.

***6.2 Twister GA scalability evaluations***

Testing environment – Quarry cluster

* 10 nodes x 8 cores
* CPU: 2.33 GB
* Memory: 16GB

While we were testing the performance test of GA implementation of Twister, we found an interesting fact which is Twister GA does not scale on the number of mappers as the other Mapreduce applications. Instead, it scales on the number of reducers. We think that is due to the fact that mappers in Twister GA just simply calculate the fitness value of each gene (count the number of ones in a gene). The computing workload is small when compared to much complex GAs. So it is a natural outcome that just increasing the number of mappers cannot significantly reduce the execution time of simple GA. However, each reducer is in charge of selection and crossover operations, which uses classic CPU approach in intensive computing. Thus, when more reducers are created as Java work threads, more computing resources are leveraged, and execution time will be reduced noticeably. In addition to that, when all cores in testing cluster are put into use, we cannot have the same execution time benefits from increasing the number of reducers because the full computing capacity of underlying system is reached. This discussion is supported by Figure 5.

Figure 5(a). Execution time plot with increasing reducers (5M input)

Figure 5(b). Execution time plot with increasing reducers (2M input)

Figure 5(c). Execution time plot with increasing reducers (1M input)

From the figures above we can tell that with the increasing input scale, the scalability on reducer numbers is becoming better and more obvious.

Besides, we also did weak scalability test on Twister GA. The result is given by Figure 6. As it illustrated, the weak scalability is not perfect. There’s constant increasing on execution time with the increasing input scale. We think it’s because of the sequential part on client of Twister GA. But the scale trend is obvious in Figure 6. After the reducer number is greater than 80, the scalability becomes worse because in our cluster we just have 80 cores available. This observation is the same as what we mentioned in discussion of Figure 5.

Figure 6. Twister GA weak scalability (1000 genes per reducer)

1. **Related Work**

Such experiment is done in University of Illinois Urbana-Champaign. They implemented Complex and Simple GA algorithms on Hadoop. They describe an algorithm design and implementation of simple and compact GAs on Hadoop, an open source implementation of MapReduce. Their experiments demonstrate the convergence and scalability up to 105 and 108 variable problems respectively. We will be focusing their Simple GA approach which is closely related to our paper.

They used the OneMax problem as their Simple GA approach as well. Authors implemented this problem on Hadoop, and ran it on 52 nodes Hadoop cluster. Each nodes run a two dual Intel Quad Cored, 16GB RAM and 2TB HDDs. The nodes are integrated into a Distributed File System which yields a storage space of 2 \* (52/3) = 34.6 TB. Each node can run 5 mappers and 3 reducers in parallel. Some of the nodes can sometimes slow down due to disk contention, disk traffic or extreme computational loads. The node that finishes first writes the output and other jobs are killed.

Each experiment has a GA population of n log n, and n is the number of variables. Their Simple GA Experiments resulted as :

1. **Convergence Analysis:** Monitor the number of bits set to 1 by the GA for a 104 variables ONEMAX problem. GA converges in 220 iterations; each iteration takes 149 seconds on average.
2. **Scalability with constant load per node:** 1000 variables are assigned per mapper. Iteration time stabilizes at 75 seconds. This points that as you add more resources, increasing the problem size does not change the iteration time. Overall map capacity is 5 \* 52 = 260, and around 250 mappers the time per iteration increases due to lack of resources to accommodate any more mappers.
3. **Scalability with constant overall load:** 50 000 variables per mapper but the number of mappers is increased. The time per iteration decreases as more mappers are added. If you keep the problem size fixed, adding more resources decreases the iteration time. 250 mappers is again the limit, and iteration time increases slightly after that. Amdahl’s law introduced by Hadoop overhead bounds the overall speedup.
4. **Scalability with increasing the problem size:** All resources are utilized but the number of variables is increased. The implementation scales up to 105 variables, keeping the population set to n log n. If more nodes are added, a large problem size can be processed. Iteration time increases sharply at n = 105 variables, as the population increases super-linearly (n log n), amounting to more than 16 million individuals.
5. **Discussions**

According to our experiments, Hadoop and Twister have distinctive characteristics. Performance is a big issue for all systems and in our problem Hadoop’s performance of Genetic Algorithms was lower than Twister’s. However, Hadoop’s programmability is straightforward because the existence of HDFS and it is not easy to make a mistake so this is an advantage for Hadoop programmers, diversely to be able to implement GA on Twister you must have a clear understanding about what is static data and what is the data flow of dynamic data. On the other hand, regardless of the strengths and weaknesses of the systems, GAs’ iterative nature was one step closer the Twister’s approach, and the results proved this anticipation.

1. **Conclusion and Future work**

Based on our experiments, Mapreduce is a good method for genetic algorithms. Especially iterative Mapreduce framework Twiser serves a better environment for genetic algorithm because its architecture is optimized for iterative applications. However, even though Hadoop is capable of doing iterative application, it is very hard to implement the application with high performance because it depends on HDFS, which involves vast number of hard disk I/O.

Even though many Mapreduce applications scale on the number of mappers, simple GA on Twister demonstrates the trait of scaling on the number of reducers. This implies that we must set the number of mappers and reducers based on the characters of applications, but not to be stick to manuals.

Twister GA’s weak scalability is not perfect owing to the sequential part, so the Amdahl's Law constrains the speed-up and scalability of Twister GA. Thus, for the further improvements of Twister GAs scalability, we considered using parallelism technique such as multiply threads to speed up the sequential part of Twister GA. In addition, we created initial population in the client. In this way, the capacity of client’s memory is the bottleneck of scalability. Therefore, a possible future work might be how to generate the initial population to overcome the bottleneck.

**REFERENCES :**

[1] Frnti, P., Kivijrvi, J., Kaukoranta, T., and Nevalainen, O. (1997). Genetic algorithms for large scale clustering problems. Compute. J, 40:547-554.

[2] Gallagher, K. and Sambridge, M. (1994). Genetic Algorithms: a powerful tool for large-scale nonlinear optimization problems. Computers & Geosciences archive., 20(7-8):1229-1236.

[3] 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. Proceeeding of the 38TH IEEE Conference on Decision and Control, 3:2288-2293

[4] E. Alba and J. M. Troya, A Survey of Parallel Distributed Genetic Algorithms, Complexity 4(1999), 31-52.

[5] Cantu-Paz, E. (2000). Efficient and Accurate Parallel Genetic Algorithms. Springer

[6] Abhishek Verma, Xavier Lora, David E. Goldberg, Roy H. Campbell (2009) Scaling Simple and Compact Genetic Algorithms using Mapreduce. IlliGAL Report No. 2009001

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

[8] 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.

[9] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung-Hee Bae, Judy Qiu, Geoffrey Fox (2010). [Twister: A Runtime for Iterative Mapreduce](http://www.iterativemapreduce.org/hpdc-camera-ready-submission.pdf)," The First International Workshop on Mapreduce and its Applications (MAPREDUCE'10) - HPDC