A Major Project Report On

Parallelizing Genetic Algorithm on Map Reduce Architecture

Submitted in partial fulfilment of the requirements

for the award of the degree of

MASTER OF TECHNOLOGY

IN

SOFTWARE ENGINEERING

By

Satyam Kashyap

(2K13/CSE/22)

Under the guidance of

Dr. Kapil Sharma

Asst. Professor

Department of Computer Science

Delhi Technological University, Delhi

Department of Computer Engineering

Delhi Technological University, Delhi

2012-2014

DELHI TECHNOLOGICAL UNIVERSITY

CERTIFICATE

This is to certify that the project report entitled **Parallelizing Genetic Algorithm on Map Reduce Architecture** is a bona fide record of work carried out by **Satyam Kashyap** (2K13/CSE/22) under my guidance and supervision, during the academic session 2013-2015 in partial fulfilment of the requirement for the degree of Master of Technology in Computer Science from Delhi Technological University, Delhi.

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University/Institute for the award of any Degree or Diploma.

Dr. Kapil Sharma
Asst. Professor
Department of Computer Science
Delhi Technological University
Delhi

DELHI TECHNOLOGICAL UNIVERSITY

ACKNOWLEDGEMENTS

I feel immense pleasure to express my heartfelt gratitude to **Dr. Kapil Sharma** for his constant and consistent inspiring guidance and utmost co-operation at every stage which culminated in successful completion of my research work.

I also would like to thank the faculty of Computer Science Department, DTU for their kind advice and help from time to time.

I owe my profound gratitude to my family which has been a constant source of inspiration and support.

Satyam Kashyap

Roll No. 2K13/CSE/22

Index

Abstract

1. Chapter 1- Introduction

2. Chapter 2- Evolutionary Algorithms and H.D.F.S.
   - 2.1 Genetic Algorithms (Simple)
   - 2.2 Genetic Algorithm 2 (Modified)
   - 2.3 Genetic Algorithm 3 (Extension)
   - 2.4 H.D.F.S.

3. Chapter 3- Map Reduce
   - 3.1 Map Reduce for Simple GA
   - 3.2 Map reduce for Modified GA

4. Chapter 4- Analysis and Evaluation
   - 4.1 GA1(Simple)
   - 4.2 GA2(Modified)
   - 4.3 GA3(Extended)

5. Chapter 5- Related Works

6. Chapter 6- Conclusion

References

Abstract

Data-intensive computation has evolved as a key component of processing large volumes of data taking advantage of massive parallelism. This computing frameworks have showcased how processing of terabytes to petabytes of raw data can be attained routinely with ease. However, Not much efforts have been done on exploring effects of evolutionary algorithms on data intensive processes. Evolutionary Algorithms like can be merged with other algorithms(presently in use) to produce better and more accurate results . Here we have presented a detailed step-by-step explanation of evolutionary computation algorithms and how they can be translated into the Map Reduce Architecture. Results have show that (1) Hadoop Architecture along with evolutionary algorithms yields better results than other contemporary algorithm in use especially in case when problem is very large and hence proves to be an excellent choice, and (2) thanks to inherent parallel processing feature of evolutionary algorithms due to which transparent linear speedups are attainable without changing the underlying data-intensive flow.

Chapter 1
Introduction

The data surge which is happening across the world in different domains of technology is forcing to rethinking of methodologies to manipulate and process large volumes of data or Big Data a modern term for the same. Most of the computing frameworks that deals with data-intensive computing [30, 7] have a common characteristic that is processing in data flow manner. Execution and nature of parallelism is derived by availability of data.

The tremendous growth of internet has brought together researchers from all disciplines of science to deal with volumes of data which can be converted to create valuable information where it seems that the only viable path is utilization of data-intensive frameworks [41, 4, 10, 29]. Though many big organizations are working on research and development of parallelizing evolutionary computation algorithms[5, 1] yet there has been very little real research work that has been done so far [25].Parallelism is inherent characteristic of evolutionary algorithms which makes them one of the optimal candidates for processing of large volumes of data [5].

Moreover, we will outline in this thesis the evolutionary algorithm(Genetic Algorithm) which serves best for this purpose and its inherent need to deal with large volumes of data, irrespective of if it takes the form of samples of a probabilistic distribution or populations of individuals in either case it will be greatly benefitted with computation model which is data intensive.

In this thesis further we will examine the usage of Yahoo!'s Hadoop model and its Map-Reduce Architecture along with implementation. This Architecture is inspired by the map and reduce methodologies both of these are primitives which are part of functional languages, Later Google proposed the Map-Reduce [8] concept which is easy to implement and empowers user to easily develop and design large distributed applications with ease.

In the proposed model computation is done though key/value pairs as input set, and produces corresponding output key/value pairs with respect to every input key value pair. Here in Map-Reduce two basic functions of the library are used that is Map and Reduce. Map in the function written by user and hence maps function can be modified a number of times according to the requirement of algorithm. Input pairs are fed into Map function which produces a set of

key/value pairs while these key value pairs are intermediate and will be modified in later stages of Map-Reduce framework. All the intermediate values created in former step will be grouped together to form Intermediate Key 1 which is now ready fed to Reduce function. Intermediate key I and a set of corresponding values are accepted by Reduce function which is also written by user.

Now merger of both of them will take place, together these values forms as smaller set of values as possible. The iterator delivers the intermediate values to reduce function which is created by user. Through this the model handles values and their list which are too large to fit into the RAM or main memory otherwise.

$$map(k1, v1) \rightarrow list(k2, v2)$$

$$reduce(k2, list(v2)) \rightarrow list(v3)$$

Automatically partitioning of the input data is done and data is divided into a set of M number of splits, Now the Map function invocations are sent and distributed across multiple machines which makes use of this partitioning. Due to the partitioning of data the input splits can be processed on different machines in parallel. Invocations of Reduce function are sent and distributed by partitioning the intermediate key space into R pieces. Hash-key partitioning function is used in the default Hadoop configuration which is which is hash(key)%R (which we can override according to our needs).User can define number of Partitions (R) and partition. Figure 1.1 shows data flow in Map-Reduce framework along with other important components of Map-Reduce. For making fault tolerant and data management scalable Map Reduce framework can be accompanied with distributed file system like GFS, though through use of HDFS we need not use GFS at all.

Here in this thesis we have selected Genetic Algorithm and developed its equivalent Map-Reduce implementations to demonstrate the benefits of the evolutionary computation and such approaches. We have paid special attention that the properties of these algorithms maintained and the fundamental mechanics should not be altered.

The three algorithms transformed were: a simple genetic algorithm [13, 14], the genetic algorithm 2 [18], and genetic algorithm 3 [19]. We will show how a simple genetic algorithm [13, 14] can be remodelled into Hadoop's MapReduce approach using data intensive computing. Reviews of following will be done (1) Steps involved in transformation of original algorithm to the algorithm which can cater data intensive framework , (2) Development and designing of data intensive components which can will yield optimal benefit in data driven approach, and (3) hence analysis of the achieved. The second example, the genetic algorithm 2 which is sibling of the simple genetic algorithm here we focus on how Hadoop's Map-Reduce modelling can help scaling it being a clear competitor of traditional in use high performance computing version.

It is worthy to notice that all of these algorithms have different nature and profiles. For example the simple genetic algorithm is better in dealing with large populations, it can tackle very huge amount of data problems, but here operators in use are pretty easy and straight forward. The 2nd version i.e. genetic algorithm 2 in comparison with Genetic Algorithm 1 is more memory efficient though it is required that simple probability distributions should be appropriately updated

Lastly, when you scale your problem size the genetic algorithm3 is optimal as it can deal with a very large population and for which a detailed model is required, to achieve the probability distribution set as target. The massive parallel data driven execution will be focused that allows users to get optimal performance in modern times of multi core processors— which has created a window to cater problems of even the scale of peta bytes— without modifying the fundamental structure of evolutionary algorithm.

In the remaining of this thesis content organization is as follows: Chapter 2 is the introduction to the three evolutionary algorithms which are genetic based. In our experimentation we will use 2 frameworks introduced above:

a) Simple genetic algorithm

b) Genetic algorithm2(Modified)

c) Genetic algorithm3(Hybrid)

In Chapter 3 all of the above algorithms will be implemented in parlance with Data intensive framework.Chapter 4 is about the presentation of analysis and evaluation of results achieved by using the data intensive executions displaying that scalability is only constrained by the availability of resources while easily achievable part is its speedup. Finally some related work and future work possible is mentioned in Chapter 5 and Chapter 6 presents conclusions of the thesis.

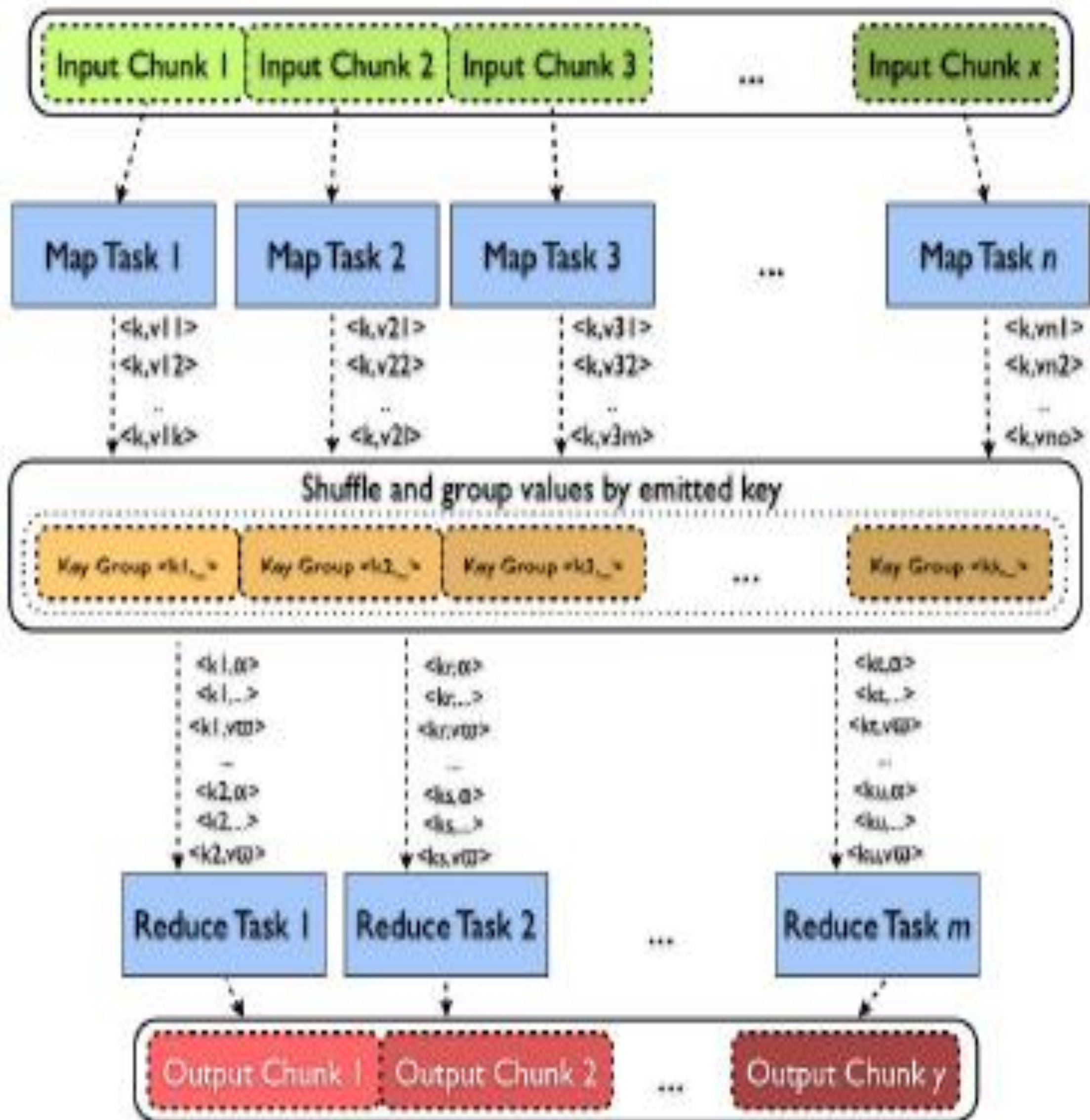


Figure 1.1

Map-Reduce Architecture

Chapter 2

Related Works

There are many models like fine grain [28], coarsed grained [24] and distributed models [23] which have been proposed for implementation of parallel GAs. Message Passing Interface (MPI) was used traditionally for parallel GAs Implementation. Problem with this framework is that MPIs don't scale very well on clusters where failure is the normal and not an rare phenomenon which it should be. In MPI failure of a node in a cluster results in failure of whole program and hence the whole program has to be restarted. When we are dealing with a large cluster iyt is more likely that we will get failures more often, to counter which efficient fault tolerance mechanism should be used. Hence, User is forced to opt for complex checkpoint methodology .

MapReduce [8] is a model which empowers the users to develop large-scale distributed applications with ease. An open source implementation of Map-Reduce mode is Hadoop. Many other implementations of Map-Reduce are available in market to facilitate other architectures as well like CGL Map-Reduce for steaming applications and Phoenix for multi-core architecture.

Only sincere attempt made so far to the best of our knowledge is MRPGA ,Which tries to combine Gas and Map-Reduce. However, According to them it is not possible to express Gas in terms of Map-reduce, Their implementation changes the very structure of Mapreduce which is never desired.. There are few shortcomings in their approach:

Firstly, fitness evaluation of genetic algorithm is done by mapper function and function of reduce here is to do local and global selection. Though many important functions are handled by single coordinator these important functions are Mutation, evaluation, cross over and convergence criteria. Due to inherent serial components scalability in their approach can not exceed beyond 32 nodes. Secondly, extension which they have proposed need not be implemented as traditional Map-Reduce have that property in built .Combiner in Map-reduce framework has same functionality as of local reduce so it is redundant.

Our approach is different, Our approach isto accommodate GAs into Map-Reduce framework rather than changing the very structure to Map-Reduce to make GAs fit into. We have implemented GAs in Hadoop, whichs is de-facto standard MapReduce implementation and is

used in commercial purpose in industries. Further we have talked about extension on GAs to multiprocessor architectures like NUMA and Phoenix and GAs inherently supports parallelizing feature and less efforts are required to manifest them.

Chapter 3

Evolutionary Algorithms(Genetic) and H.D.F.S.

In this chapter we will present the simple genetic algorithm, followed by genetic Algorithm1 and genetic algorithm 2 in detail.

3.1 Genetic Algorithms

Simple genetic algorithms [13, 14], one of the most simple forms of Genetic Algorithms, it use uses selection and recombination methodology of GAs. The data intensive approach of this basic algorithm is summarized as follows:

1. Random initialization of the individuals of population..
2. Evaluation of fitness value of each individual takes place.
3. Optimal or better solutions are selected by using S-wise tournament and it should be without replacement as after this process we will get optimal solution from pool of possible solutions
4. New population of individuals in created using crossover and recombination. Here we are using uniform crossover.
5. Fitness evaluation takes place for each individual we got in offspring population
6. Step No.3 to Step No.5 are repeated until optimal result is found.

3.2 Genetic Algorithm 2

This is modified genetic algorithm [18] and it is most simplest distribution algorithms for estimation (EDAs) [33, 22]. Similar to other EDAs, this GA replaces commonly used variation operators of genetic algorithms. Here in this algorithm we build a probabilistic model which can yield more promising solutions and with sampling the model so that new candidate solutions can be generated. We are assuming a pc=1.05,where pc is crossover probability. Probabilistic model uses vector of probabilities to represent the population and hence implicitly each gene is assumed) to be not dependent or independent of other genes. Actually, vector element represents the ratio of ones for every gene position.

The uses of probability vectors monitor more searches by generation of new candidate solutions according to the values of frequency variable by variable . The genetic algorithm2 comprises of the following steps:

1. Initialization: The population is generally initialized with the random individuals in simple Genetic algorithms but in genetic algorithm 2, initially a probability vector are set to 0.5. On the other hand ,other initializing processes can also be used in a direct manner.
2. Model sampling: Two candidate solutions are generated by sampling of teach of he probability vector. The procedure is analogous to uniform crossover in simple Genetic algorithms.
3. Evaluation: The appropriateness or the quality of each individual are measured and calculated.
4. Selection: Similar to traditional genetic algorithms, compact genetic algorithm is a selection's scheme, as only the better individual is allowed to effect the resulting generation of candidate solutions. The basic clue is that a "survival-of-the-fittest" mechanism is used to bias the generation of new individuals. We typically use tournament selection [16] in genetic algorithm 2.
5. Probabilistic model update: Afterwards selection, the percentage of winning alleles is augmented by 1/n. Take a note that the probabilities of the genes that are different between the two competitors are updated only.

The probabilistic model of GA2 is analogous to those used in population-based incremental learning (PBIL)[2, 3] and the unilabiate marginal distribution algorithm (UMDA) [32, 31].

As an alternative of shifting of the vector components proportionately to the distance from either 0 or 1, each components of the vector is updated by shifting its value by the input of a single individual to the whole frequency assuming a particular population proportions.

Moreover, GA2 considerably decreases the memory necessities when paralleled with simple genetic algorithms and PBIL. Whereas the simple Genetic algorithms required to store in the bits, GA2 merely needs to keep the proportion of ones, a finite set of n numbers that can be stored in log2 n for each of the ` gene positions. With PBIL'supdate rule, an element of the probability vector can have any arbitrary precision, and the number of values that can be stored in an element of the vector is not finite.

In another place, it has been revealed that GA2 is operationally equivalent to the order-one behavior of simple genetic algorithm with stable state selection and uniform crossover [18]. Therefore, the theory of simple genetic algorithms can be directly used to estimate the factors and behavior of the GA2. For determining the parameter we can use an approximate procedure of the gambler's ruin population-sizing2 model.

3.3 Genetic Algorithm 3

The extended compact genetic algorithm (GA3), is based on a crucial idea of the selection of a good probability distribution. It is corresponding to the linkage learning [19]. The amount of a good distribution is quantified based on minimum description length (MDL) models. The vital theory MDL models is that the given all things are equal, simple distributions are superior than the complex ones. The MDL constraints penalizes both inaccurate and complex models, thereby leading to an optimal probability distribution. The probability distribution used in GA3 is a class of probability models called marginal product models (MPMs).

MPMs are formed as a product of marginal distributions on a partition of the genes. MPMs also enable a direct linkage map with every partition untying tightly linked genes.

1. Initialization of the population with random individuals.

2. Evaluation of the fitness value of the individuals.

3. Good solutions selection by by means of s-wise tournament selection without replacement [16].

4. Building of the probabilistic model: In _-GA3, both the structure of the model as well as the parameters of the models are examined. A greedy search is used to search for the model of the selected individuals in the population.

5. Creation of new individuals by sampling of the probabilistic model.

6. Evaluation of the fitness value of all offspring.

7. Repeatition of steps 3–6 until some convergence criteria are met.

The materialistic exploration experimental used in _-GA3 starts with a simple model assuming all the variables to be independent and consecutively merges subsets until the MDL metric no longer progresses.

As soon as the model is built and the marginal probabilities are computed, a new population is produced based on the optimal MPM as follows, population of size n(1 − pc) where pc implies the crossover probability which is occupied by the best individuals in the current population. The rest n • pc individuals are produced by random selection of subsets from the current individuals rendering from the probabilities of the subsets as designed in the model.

One of the important parameters which determines the success of GA3 is the population size. Analytical models have been established for prediction of the population-sizing and the scalability of GA3 [36].

The models predict that the population size required to solve a problem with m building blocks of size k with a

failure rate of _ = 1/m is given by

$$n \propto \chi^k \left(\frac{\sigma_{BB}^2}{d^2} \right) m \log m,$$

3.4 Hadoop Distributed File System

3.4.1 Introduction

Hadoop is an Apache project; all components are available via the Apache open source license. Yahoo! has developed and contributed to 80% of the core of Hadoop (HDFS and MapReduce). HBase was originally developed at Powerset, now a department at Microsoft. Hive [15] was originated and develdeveloped at Facebook. Pig [4], ZooKeeper [6], and Chukwa were originated and developed at Yahoo! Avro was originated at Yahoo! and is being co-developed with Cloudera. HDFS is the file system component of Hadoop. While the interface to HDFS is patterned after the UNIX file system,faithfulness to standards was sacrificed in favor of improved performance for the applications at hand. HDFS stores file system metadata and application data separately. As in other distributed file systems, like PVFS [2][14], Lustre [7] and GFS [5][8], HDFS stores metadata on a dedicated server, called the NameNode. Application data are stored on other servers called DataNodes. All servers are fully connected and communicate with each other using TCP-based protocols. Unlike Lustre and PVFS, the DataNodes in HDFS do not use data protection mechanisms such as RAID to make the data durable. Instead, like GFS, the file content is replicated on multiple DataNodes for reliability. While ensuring data durability, this strategy has the added advantage that data transfer bandwidth is multiplied, and there are more opportunities for locating
computation near the needed data. Several distributed file systems have or are exploring truly distributed implementations of the namespace. Ceph [17] has a
cluster of namespace servers (MDS) and uses a dynamic subtree partitioning algorithm in order to map the namespace tree to MDSs evenly. GFS is also evolving into a distributed namespace implementation [8]. The new GFS will have hundreds of
namespace servers (masters) with 100 million files per master. Lustre [7] has an implementation of clustered namespace on its roadmap for Lustre 2.2 release. The intent is to stripe a directory over multiple metadata servers (MDS), each of which contains a disjoint portion of the namespace. A file is assigned to a particular MDS using a hash function on the file name.

3.4.2 Architecture

NameNode

The HDFS namespace is a hierarchy of files and directories. Files and directories are represented on the NameNode by which record attributes like permissions, modification and access times, namespace and disk space quotas. The file content is split into large blocks (typically 128 megabytes, but user selectable file-by-file) and each block of the file is independently
replicated at multiple DataNodes (typically three, but user selectable file-by-file). The NameNode maintains the namespace tree and the mapping of file blocks to DataNodes (the physical location of file data). An HDFS client wanting to read a file first contacts the NameNode for the locations of data blocks comprising the file and then reads block contents from the DataNode closest to the client. When writing data, the client requests the NameNode to nominate a suite of three DataNodes to host the block replicas. The client then writes data to the DataNodes in a pipeline fashion. The current design has a single NameNode for each cluster. The cluster can have thousands of DataNodes and tens of thousands of HDFS clients per cluster, as each DataNode may execute multiple application tasks concurrently. HDFS keeps the entire namespace in RAM. The inode data and the list of blocks belonging to each file comprise the metadata of the name system called the image. The persistent record of the image stored in the local host's native files system is called a checkpoint. The NameNode also stores the modification log of the image called the journal in the local host's native file system. For improved durability, redundant copies of the checkpoint and journal can be made at other servers. During restarts the NameNode restores the namespace by reading the namespace and replaying the journal. The locations of block replicas may change over time and are not part of the persistent checkpoint.

DataNodes

Each block replica on a DataNode is represented by two files in the local host's native file system. The first file contains the data itself and the second file is block's metadata including

checksums for the block data and the block's generation stamp. The size of the data file equals the actual length of the block and does not require extra space to round it up to the nominal block size as in traditional file systems. Thus, if a block is half full it needs only half of the space of the full block on the local drive. During startup each DataNode connects to the NameNode and performs a handshake. The purpose of the handshake is to verify the namespace ID and the software version of the DataNode. If either does not match that of the NameNode the DataNode automatically shuts down.The namespace ID is assigned to the file system instance when it is formatted. The namespace ID is persistently stored on all nodes of the cluster. Nodes with a different namespace

ID will not be able to join the cluster, thus preserving the integrity of the file system. The consistency of software versions is important because incompatible version may cause data corruption or loss, and on large clusters of thousands of machines it is easy to overlook nodes that did not shut down properly prior to the software upgrade or were not available during the upgrade. A DataNode that is newly initialized and without any namespace ID is permitted to join the cluster and receive the cluster's namespace ID. After the handshake the DataNode registers with the NameNode. DataNodes persistently store their unique storage IDs. The storage ID is an internal identifier of the DataNode, which makes it recognizable even if it is restarted with a different IP address or port. The storage ID is assigned to the DataNode when it registers with the NameNode for the first time and never changes after that. A DataNode identifies block replicas in its possession to the NameNode by sending a block report. A block report contains the block id, the generation stamp and the length for each block replica the server hosts. The first block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour and provide the NameNode with an up-todate view of where block replicas are located on the cluster. During normal operation DataNodes send heartbeats to the NameNode to confirm that the DataNode is operating and the block replicas it hosts are available. The default heartbeat interval

is three seconds. If the NameNode does not receive a heartbeat from a DataNode in ten minutes the NameNode considers the DataNode to be out of service and the block replicas hosted by that DataNode to be unavailable. The NameNode then schedules creation of new replicas of those blocks on other DataNodes. Heartbeats from a DataNode also carry information about total storage capacity, fraction of storage in use, and the number of data transfers currently in

progress. These statistics are used for the NameNode's space allocation and load balancing decisions. The NameNode does not directly call DataNodes. It uses replies to heartbeats to send instructions to the DataNodes. The instructions include commands to:

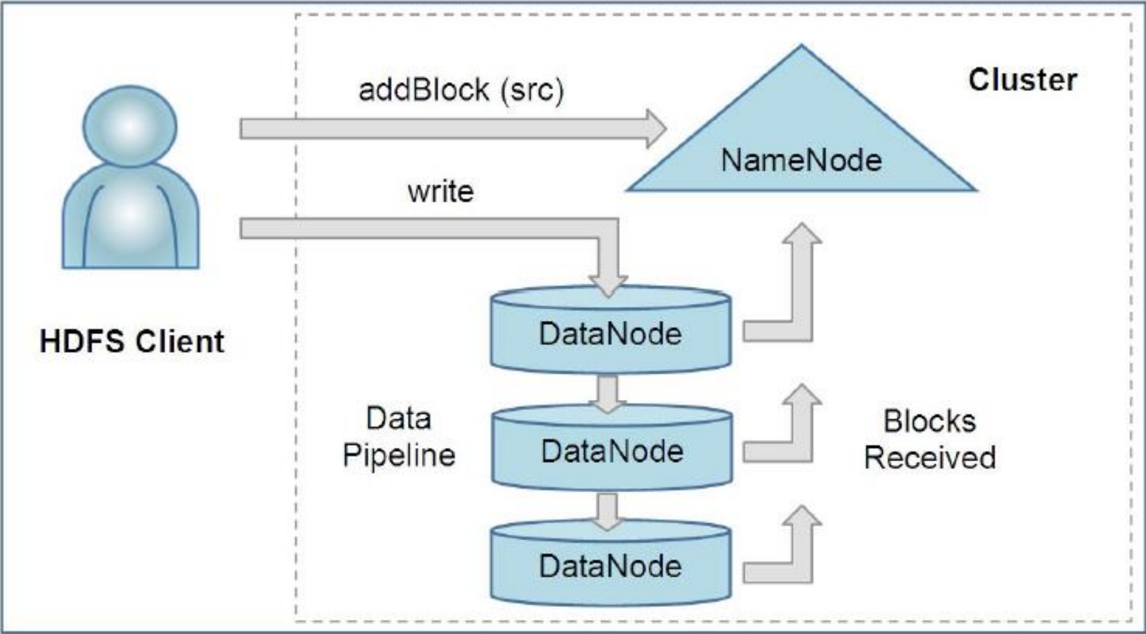
- replicate blocks to other nodes;
- remove local block replicas;
- re-register or to shut down the node;
- send an immediate block report.

These commands are important for maintaining the overall system integrity and therefore it is critical to keep heartbeats frequent even on big clusters. The NameNode can process thousands of heartbeats per second without affecting other NameNode operations.

HDFS Client

User applications access the file system using the HDFS client, a code library that exports the HDFS file system interface. Similar to most conventional file systems, HDFS supports operations to read, write and delete files, and operations to create and delete directories. The user references files and directories by paths in the namespace. The user application generally does not need to know that file system metadata and storage are on different servers, or that blocks have multiple replicas. When an application reads a file, the HDFS client first asks the NameNode for the list of DataNodes that host replicas of the blocks of the file. It then contacts a DataNode directly and requests the transfer of the desired block. When a client writes, it first asks the NameNode to choose DataNodes to host replicas of the first block of the file. The client organizes a pipeline from node-to-node and sends the data. When the first block is filled, the client requests new DataNodes to be chosen to host replicas of the next block. A new pipeline is organized, and the client sends the further bytes of the file. Each choice of DataNodes is likely to be different. The interactions among the client, the NameNode and the DataNodes are Unlike conventional file systems, HDFS provides an API that exposes the locations of a file blocks. This allows applications like the MapReduce framework to schedule a task to where the data are located, thus improving the read performance. It also allows an application to set the replication factor of a file. By default a file's replication factor is three. For critical files or files which are

accessed very often, having a higher replication factor improves their tolerance against faults and increase their read bandwidth.



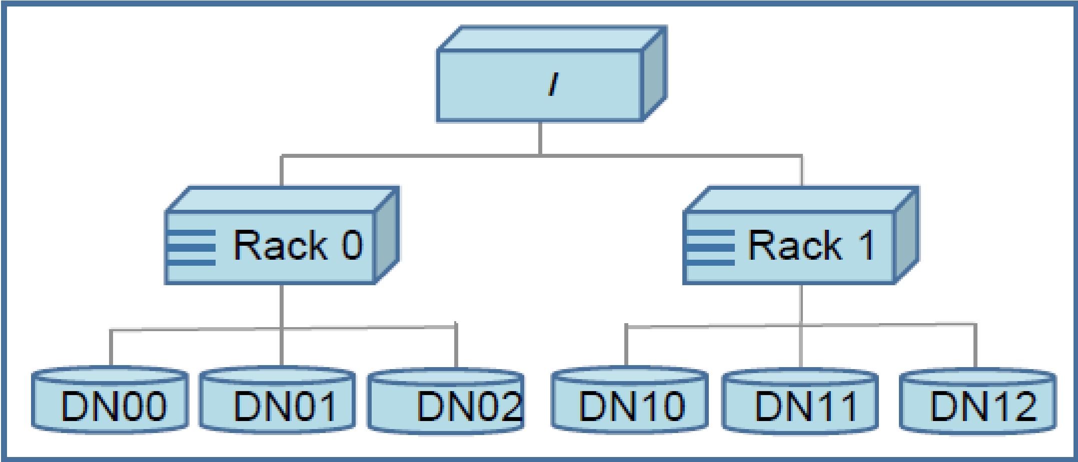


Figure 3. Cluster topology example

Image and Journal

The namespace image is the file system metadata that describes the organization of application data as directories and files. A persistent record of the image written to disk is called a checkpoint. The journal is a write-ahead commit log for changes to the file system that must be persistent. For each client-initiated transaction, the change is recorded in the journal, and the journal file is flushed and synched before the change is committed to the HDFS client. The checkpoint file is never changed by the NameNode; it is replaced in its entirety when a new checkpoint is created during restart, when requested by the administrator, or by the CheckpointNode described in the next section. During startup the NameNode initializes the namespace image from the checkpoint, and then replays changes from the journal until the image is up-to-date with the last state of the file system. A new checkpoint and empty journal are written back to the storage directories before the NameNode starts serving clients. If either the checkpoint or the journal is missing, or becomes corrupt, the namespace information will be lost partly or entirely. In order to preserve this critical information HDFS can be configured to store the checkpoint and journal in multiple storage directories. Recommended practice is to place the directories on different volumes, and for one storage directory to be on a remote NFS server. The first choice prevents loss from single volume failures, and the second choice protects against failure of the entire node. If the NameNode encounters an error writing the journal to one of the storage directories it automatically excludes that directory from the list of storage directories. The NameNode automatically shuts itself down if no storage directory is available. The NameNode is a multithreaded system and processes requests simultaneously from multiple clients. Saving a transaction to disk becomes a bottleneck since all other threads need to wait until the synchronous flush-and-sync procedure initiated by one of them is complete. In order to optimize this process the NameNode batches multiple transactions initiated by different clients. When one of the NameNode’s threads initiates a flush-and-sync operation, all transactions batched at that time are committed together. Remaining threads only need to check that their transactions have been saved and do not need to initiate a flush-and-sync operation.

Checkpoint Node

The NameNode in HDFS, in addition to its primary role serving client requests, can alternatively execute either of two other roles, either a CheckpointNode or a BackupNode. The role is specified at the node startup. The CheckpointNode periodically combines the existing checkpoint and journal to create a new checkpoint and an empty journal. The CheckpointNode usually runs on a different host from the NameNode since it has the same memory requirements as the NameNode. It downloads the current checkpoint and journal files from the NameNode, merges them locally, and returns the new checkpoint back to the NameNode. Creating periodic checkpoints is one way to protect the file system metadata. The system can start from the most recent checkpoint if all other persistent copies of the namespace image or journal are unavailable. Creating a checkpoint lets the NameNode truncate the tail of the journal when the new checkpoint is uploaded to the NameNode. HDFS clusters run for prolonged periods of time without restarts during which the journal constantly grows. If the journal grows very large, the probability of loss or corruption of the journal file increases. Also, a very large journal extends the time required to restart the NameNode. For a large cluster, it takes an hour to process a week-long journal. Good practice is to create a daily checkpoint.

BackupNode

A recently introduced feature of HDFS is the BackupNode. Like a CheckpointNode, the BackupNode is capable of creating periodic checkpoints, but in addition it maintains an inmemory,
up-to-date image of the file system namespace that is always synchronized with the state of the NameNode. The BackupNode accepts the journal stream of namespace transactions from the active NameNode, saves them to its own storage directories, and applies these transactions to its own namespace image in memory. The NameNode treats the BackupNode as a journal store the same as it treats journal files in its storage directories. If the NameNode fails, the BackupNode's image in memory and the checkpoint on disk is a record of the latest namespace state. The BackupNode can create a checkpoint without downloading checkpoint and journal files from the active NameNode, since it already has an up-to-date namespace image in its memory. This

makes the checkpoint process on the BackupNode more efficient as it only needs to save the namespace

into its local storage directories. The BackupNode can be viewed as a read-only NameNode. It contains all file system metadata information except forblock locations. It can perform all operations of the regular NameNode that do not involve modification of the namespace or knowledge of block locations. Use of a BackupNode provides the option of running the NameNode without persistent storage, delegating responsibility for the namespace state persisting to the BackupNode.

Upgrades, File System Snapshots

During software upgrades the possibility of corrupting the system due to software bugs or human mistakes increases. The purpose of creating snapshots in HDFS is to minimize potential damage to the data stored in the system during upgrades. The snapshot mechanism lets administrators persistently save the current state of the file system, so that if the upgrade results in data loss or corruption it is possible to rollback the upgrade and return HDFS to the namespace and storage state as they were at the time of the snapshot. The snapshot (only one can exist) is created at the cluster administrator's option whenever the system is started. If a snapshot is requested, the NameNode first reads the checkpoint and journal files and merges them in memory. Then it writes

the new checkpoint and the empty journal to a new location, so that the old checkpoint and journal remain unchanged. During handshake the NameNode instructs DataNodes whether to create a local snapshot. The local snapshot on the DataNode cannot be created by replicating the data files directories as this will require doubling the storage capacity of every DataNode on the cluster. Instead each DataNode creates a copy of the storage directory and hard links existing block files into it. When the DataNode removes a block it removes only the hard link, and block modifications during appends use the copy-on-write technique. Thus old block replicas remain untouched in their old directories. The cluster administrator can choose to roll back HDFS to the snapshot state when restarting the system. The NameNode recovers the checkpoint saved when the snapshot was created. DataNodes restore the previously renamed directories and initiate a background process to delete block replicas created after the snapshot was made. Having chosen

to roll back, there is no provision to roll forward. The cluster administrator can recover the storage occupied by the snapshot by commanding the system to abandon the snapshot, thus finalizing the software upgrade.

System evolution may lead to a change in the format of the NameNode's checkpoint and journal files, or in the data representation of block replica files on DataNodes. The layout version identifies the data representation formats, and is persistently stored in the NameNode's and the DataNodes' storage directories. During startup each node compares the layout version of the current software with the version stored in its storage directories and automatically converts data from older formats

to the newer ones. The conversion requires the mandatory creation of a snapshot when the system restarts with the new software layout version. HDFS does not separate layout versions for the NameNode and DataNodes because snapshot creation must be an all cluster effort rather than a node-selective event. If an upgraded NameNode due to a software bug purges its image then backing up only the namespace state still results in total data loss, as the NameNode will not recognize the blocks reported by DataNodes, and will order their deletion. Rolling back in this case will recover the metadata, but the data itself will be lost. A coordinated snapshot is required to avoid a cataclysmic destruction.

Chapter 4

Map-Reduce

In this chapter, Transformation and implementation of simple model of genetics algorithm is being done with the help of Map-Reduce along with discussion of some of the fundamentals that are necessary to be taken into account.

4.1 MapReduce SGAs

Map Reduce job is summarized separately in each repetition of genetic algorithm. The acceptance the command-line parameters by client than creation of the population and finally submission of the Map-Reduce job.

One of the simplest forms of Genetic algorithm is Selecto-recombinative genetic algorithms[13,14], which chiefly depend on the use of selection and recombination. We selected it to start with because they are presented with a minimum set of operators that will support us illustrate the formation of a data-intensive flow complement or counterpart.

4.1.1 Map

The fitness function Evaluation for the population (Steps 2 and 5) matches the Map function, which has to be computed independently of other illustrations. The Map evaluates the fitness level of the given individual, as shown in the algorithm in Algorithm 1.

Also, it preserves track of the fittest individual and lastly, writes it to a global dossier in the Hadoop Distributed File System (HDFS). The client, who has started the job, reads these values from all the mappers at the end of the Map-Reduce and checks if the convergence criteria has been fulfilled.

4.1.2 Partitioner

For selection operation of Genetic Algorithm Step 3 is locally performed on every node, to reduce selection pressure spatial constraints are artificially introduced in it. Both the reasons mentioned above results in increase on convergence time of genetic algorithm. Therefore decentralized and distributed selection operation is preferred .In Whole map reduce model the only time at which real global communication takes place is at the time of shuffle of Map-Reduce.

Figure below presents

At the final phase of Map, key/value pairs are shuffled by Map-Reduce framework via partitioner to reducers. Intermediate key/value pairs are splitted by partitioner. Now reducer is responsible for processing of given set of key/value pairs,function getPartition() returns the reducer. The Hash(key) % num is used as the default. This is used as default so that all the values corresponding to a given key converge on single reducer.

Algorithm 1 Map phase of each iteration of the GA

```
1:  MAP(key, value):
2:  individual ← INDIVIDUALREPRESENTATION(key)
3:  fitness ← CALCULATEFITNESS(individual)
4:  EMIT (individual, fitness)

5:  {Keep track of the current best}
6:  if fitness > max then
7:          max ← fitness
8:          maxInd ← individual
9:  end if

10: if all individuals have been processed then
11:         Write best individual to global file in DFS
12: end if
```

There are 2 reasons because of which genetic algorithm is not suited for it:

First is that the Hash function we are using partitions the N individuals into r number of distinct classes given by : $N0,N1, \ldots, Nr-1$ where $Ni = \{n : Hash(n) = i\}$. though individuat inside a partition is away from individuals in other partitions. Thus we can say that Hash Partitioner brings artificial spatial constraint which in lower bits. The result of which is convergence of iterations may take longer amount of time or may not converge ever.

Second reason is, as the algorithm proceeds individuals with better fitness scores start to dominate flow of algorithm . Due to which reducers will get overloaded as all the copies are sent to same reduced. Thus, as the algorithm progress distribution becomes more skewed which starts deviating from the distribution which desired to be produced in a GA i.e uniform distribution (uniform distribution can maximize use of parallelism). Finally, at the time of convergion of Gas all the individuals will be targeted to ward same reducer which will overload it and result will be more execution time per iteration.

For these above given reasons we need to override the default partitioner and provide our own partitioner, our partitoner is expected to shuffle more randomly hence share load of all the reducers

4.1.3 Reduce

Tournament selection without replacement is used in our algorithm. Among S randomly chosen individual a tournament is conducted and the winner is selected. We repeat as many times as the number of populations. Since arbitrarily choosing candidates it is equivalent to first randomly shuffling and then processing all the individuals sequentially, reduce function which we have formulated goes works sequentially on all the individuals. Individuals are buffered for the final rounds Initially, and after the window of tournament is full, two basic operation of GAs are done to bring out new populations i.e. Selection And Crossover . The Uniform Crossover operator is used When the crossover window is full. In our implementation we haveset the S to 5 and consecutively selected parents are selected among which crossover is done.

Algorithm 3 Reduce phase of each iteration of the GA

```
1: Initialize processed ← 0, tournArray [2· tSize], crossArray [cSize]

2: REDUCE(key, values):
3: while values.hasNext() do
4:         individual ← INDIVIDUALREPRESENTATION(key)
5:         fitness ← values.getValue()

6:         if processed < tSize then
7:                 {Wait for individuals to join in the tournament and put them for the last rounds}
8:                 tournArray [tSize + processed%tSize] ← individual
9:         else
10:                {Conduct tournament over past window}
11:                SELECTIONANDCROSSOVER()
12:        end if
13:        processed ← processed + 1

14:        if all individuals have been processed then
15:                {Cleanup for the last tournament windows}
16:                for k ←1 to tSize do
17:                        SELECTIONANDCROSSOVER()
18:                        processed ← processed + 1
19:                end for
20:        end if
21: end while

22: SELECTIONANDCROSSOVER:
23: crossArray[processed%cSize] ← TOURN(tournArray)
24: if (processed - tSize) % cSize = cSize - 1 then
25:        newIndividuals ← CROSSOVER(crossArray)
26:        for individual in newIndividuals do
27:                EMIT (individual, dummyFitness)
28:        end for
29: end if
```

4.1.4 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 speed-up is bounded because of this serial component.Amdahl's law states: if p is the proportion of a program that can be made parallel (i.e. benefit from parallelization), and $(1 - p)$ is the proportion that cannot be

parallelized (remains serial), then the maximumspeed-up that can be achieved by using n processors in the limit, as n tends to infinity tends to $1/(1 - p)$.

Thus, the speed up is bound by fraction of time of 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 Reducer1. We seed the pseudo-random number generator for each mapper with mapper id · current time. The bits of the variables in the individual are compactly represented in an array of long long ints 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.

4.2 MapReducing Genetic algorithm 2s

We encapsulate each iteration of the GA2 as a separate single MapReduce job. The client accepts the command-line parameters, creates the initial probability vector splits and submits the MapReduce job. Let the probability vector be P = {pi : pi = Probability of the variable(i) = 1}. Such an approach would allow us to scale in terms of the number of variables, if P is partitioned into m different partitions P1, P2, . . . , Pm where m is the number of mappers.

4.2.1 Map

Generation of the two individuals matches the Map function, which has to be computed independent of other instances. As shown in the algorithm in Algorithm 3.2.1, the Map takes a probability split Pi as input and outputs the tournament .Size individuals splits, as well as the probability split. Also, it keeps track of the number of ones in both the individuals and writes it to a global file in the Distributed File System (HDFS). All the reducers later read these values.

Chapter 5

Analysis and Evaluation

In this chapter, the conclusion of the experiments done for the evaluation of the algorithms presented in the previous chapter will be described.

We instigated the Map-Reduce algorithms on Hadoop (0.19)1 and ran it on our 16 core (8 nodes) Hadoop cluster. Each one of the node runs two dual Intel cores that is 6GB RAM and 1TB hard disks. These integrated nodes are Distributed File System (HDFS) which produces a potential single image storage space of 2*8/3 = 0.53 TB (since the replication factor of HDFS is set to 3). A meticulous explanation of these cluster setup can be found elsewhere. Each and every node can run 5 mappers and 3 reducers in parallel at the same time.

Due to disk contention, network traffic, or extreme computation loads some of the nodes despite being fully functional can be slowed down. Therefore, Hypothetical execution should be used to run the jobs assigned to these slow nodes, on idle nodes in parallel. Whichever node finishes first, output is written by that node 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.

5.1 Simple Genetic Algorithm

The One Max Problem also known as Bit Counting, is a simple problem involved in maximizing the number of ones of a bit string. Formally this problem is described as finding an string x = {x1, x2, . . . , xN}, with xi 2 {0, 1}, that maximizes the following equation:

$$F(\bar{x}) = \sum_{i=1}^{N} x_i$$

Here OneMax problem is used for evaluation and implementation of our simple genetic algorithms and in performing of the following experiments:

5.1.1 Convergence Analysis

In this experiment, we display the progress in terms of the number of bits set to 1 by the GA for a 104 variable One MAX problem. As shown in Figure 4.1, the GA converges in 220 iterations taking an average of 149 seconds per iteration.

5.1.2 Scalability and continuous load per node

In this experiment, load is kept 1,000 variables per mapper, as displayed in Figure 4.2. The time per iteration rises at first and then it stabilizes to about 75 seconds. This increases the problem size, as more resources are added which does not change the iteration time. As each and every node can run a maximum of 5 mappers making the overall map capacity as 5・52(nodes) = 260. Therefore around 250 mappers time per iteration is increased due to the lack of resources to accommodate so many mappers.

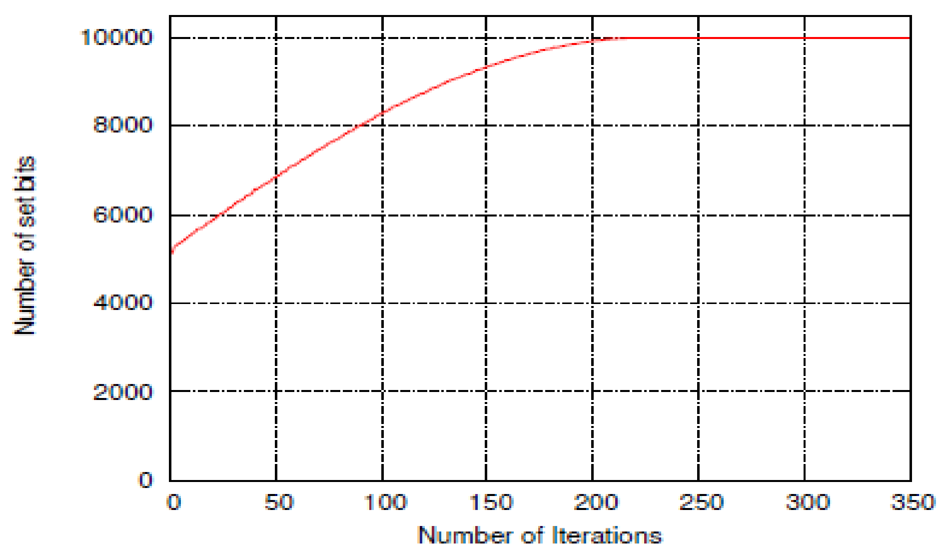


Figure 4.1: Convergence of GA for 10^4 variable ONEMAX problem

5.1.3 Scalability with constant overall load

In this experiment, the problem size is kept fixed to 50,000 variables and the number of mappers is increased.

As shown in Figure 4.3, the time per iteration decreases As more and more mappers are added, the time per iteration decreases 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.

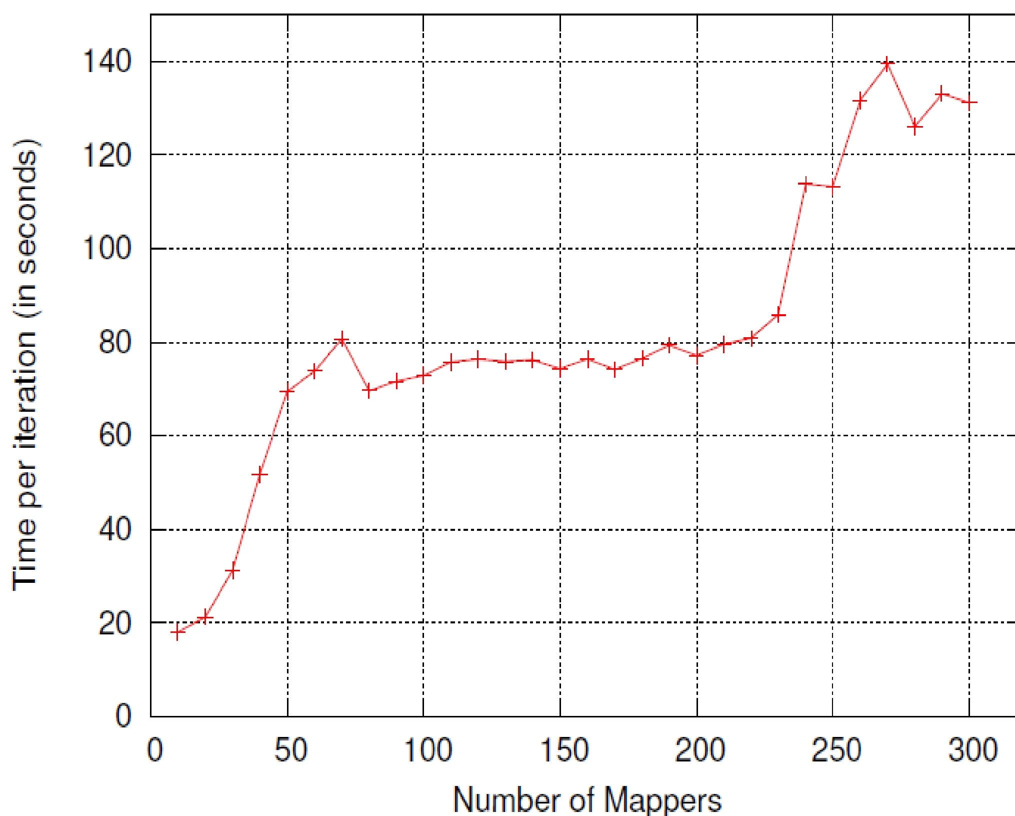


Figure 4.2: Scalability of GA with constant load per node for ONEMAX problem

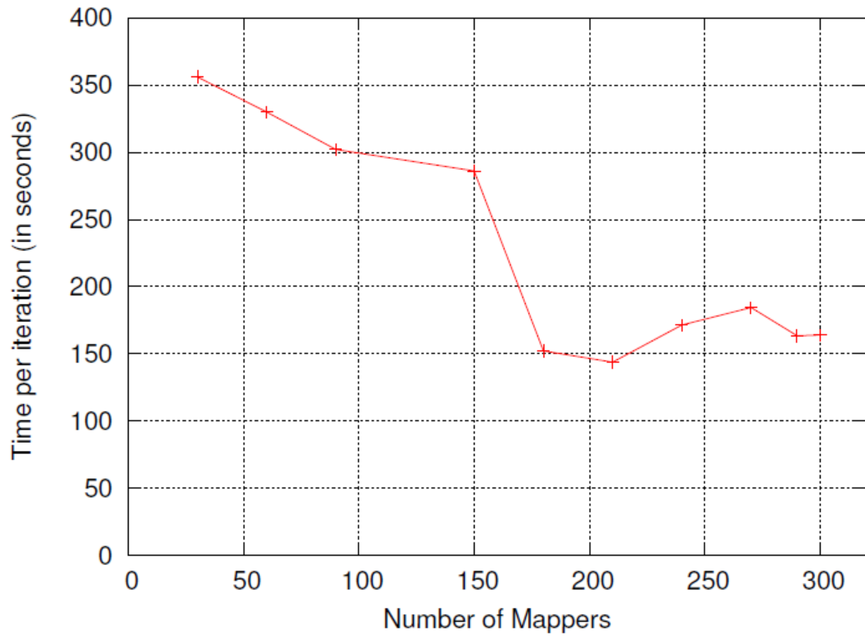


Figure 4.3: Scalability of GA for 50,000 variable ONEMAX problem with increasing number of mappers

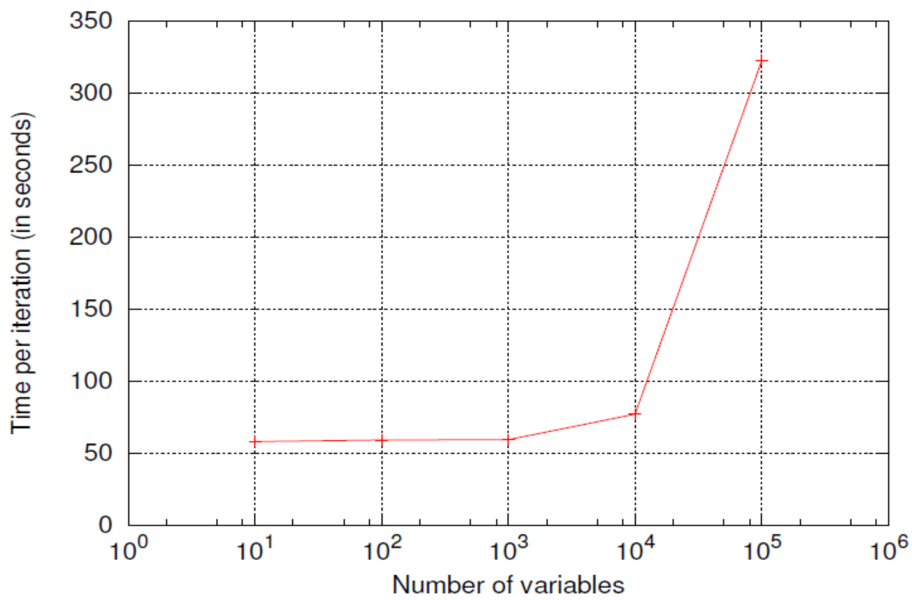


Figure 4.4: Scalability of GA for ONEMAX problem with increasing number of variables

5.1.4 Scalability with increasing the problem size

Here, maximum resources are utilized and the number of variables are also increased. As presented in Figure 4.4, our implementation scales to n = 105 variables, keeping the population set to n log n. Addition of more nodes would allow us to scale larger problem sizes. The time per iteration will be increases abruptly as the number of variables is increased to n = 105 as the population increases super-linearly (n log n), which is more than 16 million individuals.

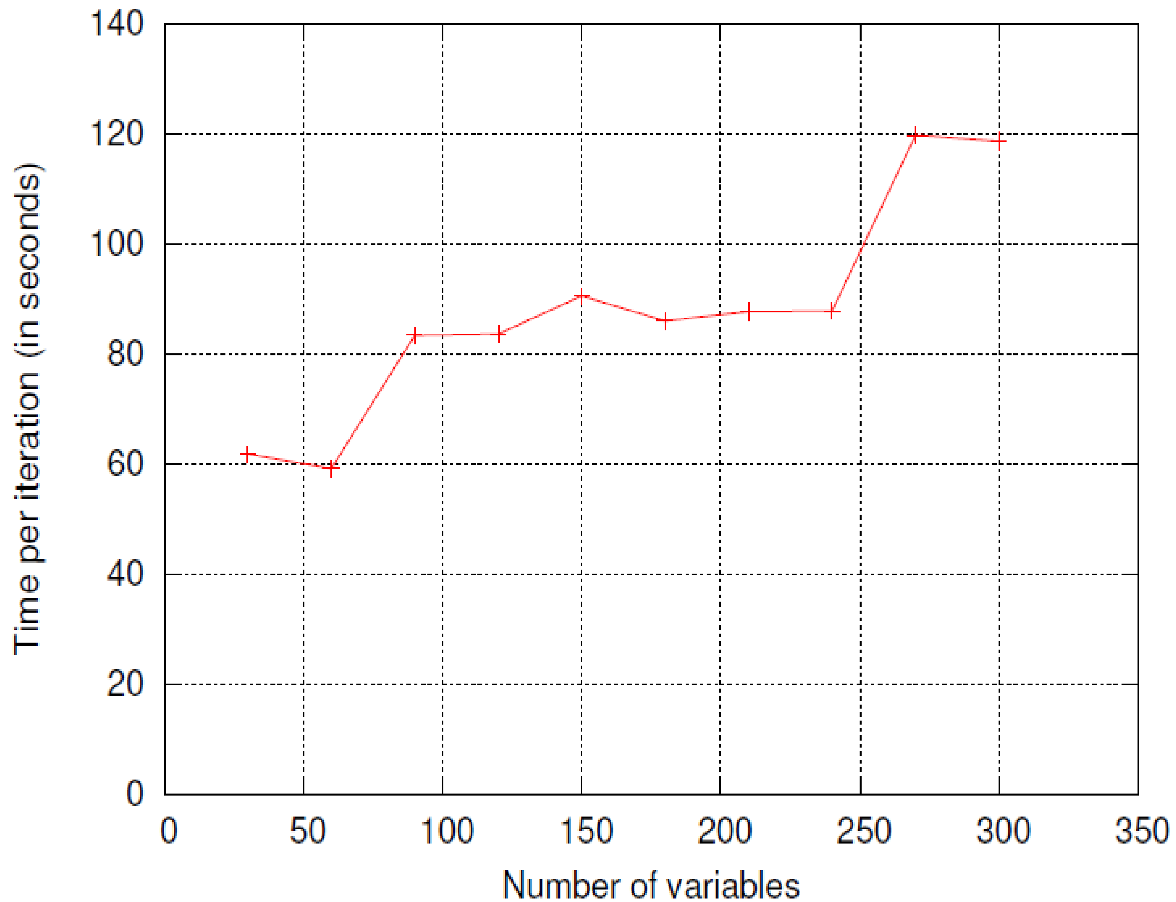


Figure 4.5

5.2 Genetic Algorithms2

For better understanding of the behavior of the Hadoop implementation of GA2, we repeat the two experiment sets which were done in the case of the Hadoop SGA implementation. For each experiment, the population for the GA2 is set to n log n where n is the number of variables. As done earlier, first the load set is kept to 200,000 variables per mapper. As shown in Figure 4.5, the time per iteration increases at first 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 · 52(nodes) = 260. Hence, around 250 mappers, the time per iteration increases due to the fact that no available resources (mapper slots) in the Hadoop framework are available. Thus, the execution must wait till mapper slots are released and the remaining portions can be executed, and the whole execution completed.

In the second set of experiments, we utilized the maximum resources and increase the number of variables. As shown in Figure 4.6, our implementation scales to n = 108 variables, keeping the population set to n log n.

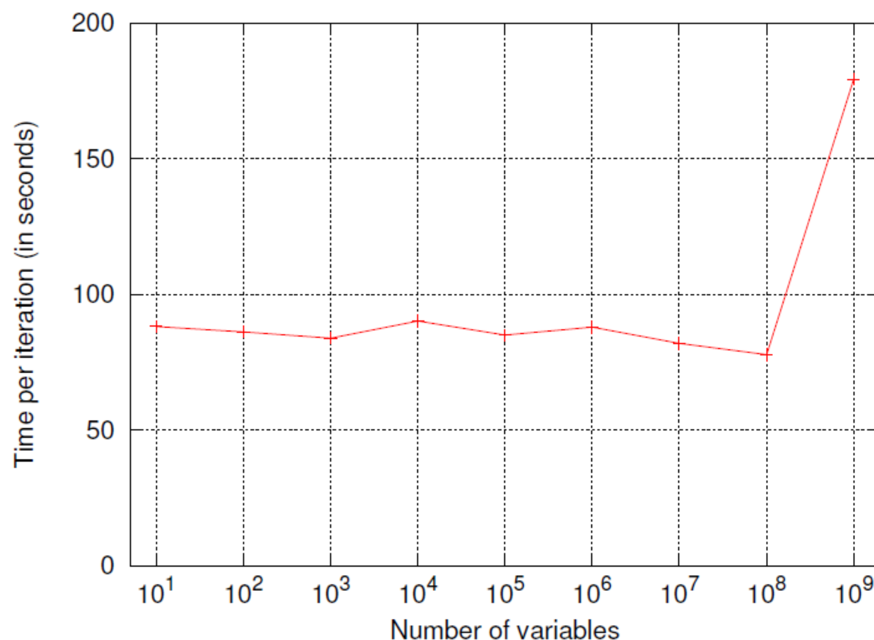


Figure 4.6: Scalability of compact genetic algorithm for OneMAX problem with increasing number of variables.

5.3 Genetic Algorithm 3

We here are reporting our results of the experiments with MapReducing GA3 algorithms.

5.3.1 Convergence

In order to confirm the accuracy of our parallel implementation of the GA3 algorithm, we ran an experiment on a problem with 16 bit variables and it is converged in three iterations for attaining the best possible fitness. We demonstrate the model building process that resulted in the last iteration in the following listing:

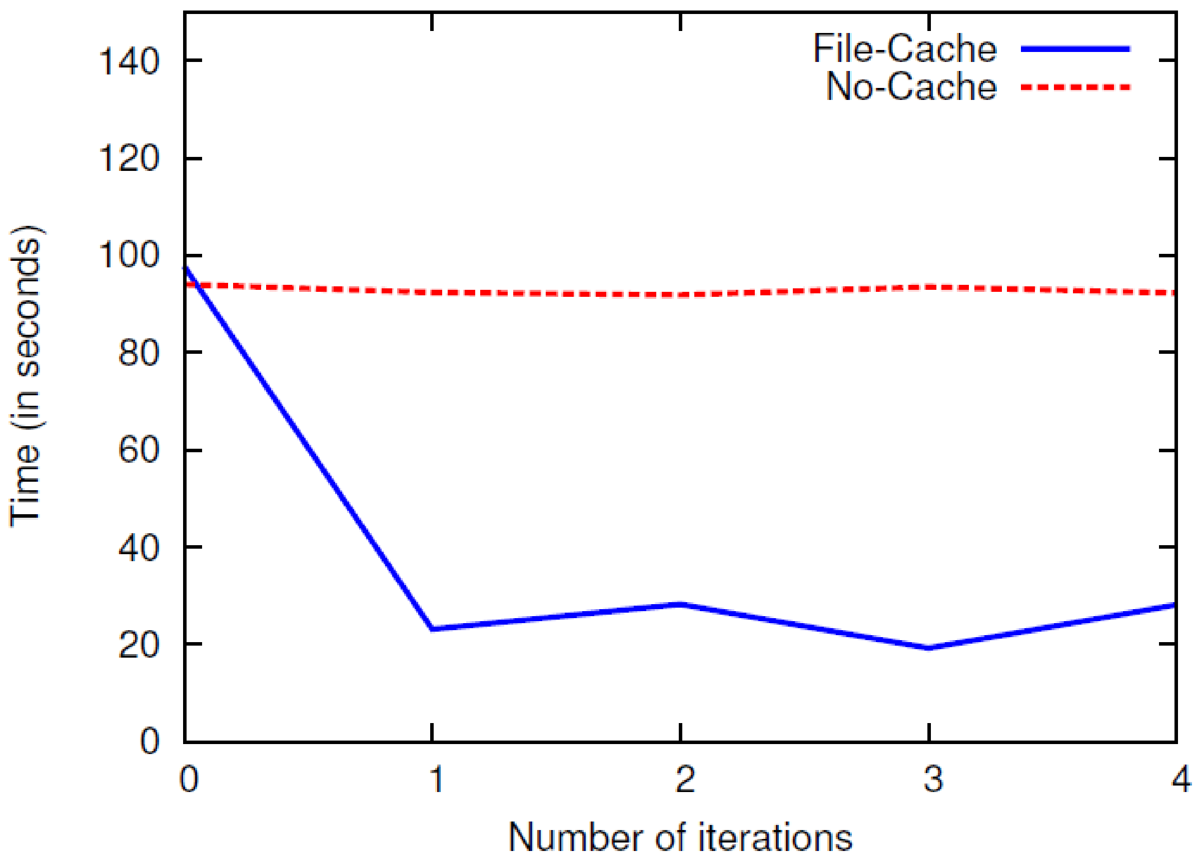


Figure 4.7: Effect of caching on the iteration time.

$$(0)\ (1)\ (2)\ (3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12)\ (13)\ (14\ 15)$$

$$(0)\ (1)\ (2\ 3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12)\ (13)\ (14\ 15)$$

$$(0)\ (1\ 2\ 3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12)\ (13)\ (14\ 15)$$

$$(0)\ (1\ 2\ 3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12)\ (13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12)\ (13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5)\ (6)\ (7)\ (8)\ (9)\ (10)\ (11)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 7)\ (6)\ (8)\ (9)\ (10)\ (11)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 6\ 7)\ (8)\ (9)\ (10)\ (11)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 6\ 7)\ (8)\ (9\ 11)\ (10)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 6\ 7)\ (8)\ (9\ 10\ 11)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 6\ 7)\ (8\ 9\ 10\ 11)\ (12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4)\ (5\ 6\ 7)\ (8\ 9\ 10\ 11\ 12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3)\ (4\ 5\ 6\ 7)\ (8\ 9\ 10\ 11\ 12\ 13\ 14\ 15)$$

$$(0\ 1\ 2\ 3\ 4\ 5\ 6\ 7)\ (8\ 9\ 10\ 11\ 12\ 13\ 14\ 15)$$

5.3.2 Caching

In this experiment, we measure the advantages of caching in the model building phase of the GA3 algorithm. In the first iteration, we compute the marginal probabilities of each building block in the map phase and the the marginal probabilities of each pair-wise combination of the building block. If we don't cache these marginal probabilities, they are computed in every iteration of the model building process. This is demonstrated in the "No-Cache" line in Figure 4.7. We can cache most of this information for the next iteration, as only the merged building block will have different marginal probabilities. This results in upto 80% lesser time per iteration, as is demonstrated in the "File-cache" line in the same figure.

5.3.3 Scaling the model building with problem size

In this experiment, we analyze the average time per iteration in the model building process for different problem sizes. Our results show that for problem sizes up to 128, the start-up overhead of the MapReduce results in similar execution times for the no-cache and file-cache versions as shown in Figure 4.8. The difference becomes more prominent for larger problem sizes. Our implementation scales up to 1024 bit variable problems. We found that beyond this value, the memory overhead of maintaining marginal probabilities for each pair-wise merge of building blocks becomes the bottleneck.

5.3.4 Scaling the model building with number of mappers

In this Experiment we can see effect of increasing number of mappers on the scalability. Figure 4.9 shows as the number of mappers are increased relative effect on average time per iteration. It can be observed from the figure that as the number of mappers is small, load per mapper is high

and hence time per iteration is also high correspondingly. And when the same amount of work is distributed on several number of machine, load on per mapper reduces as the result time of execution per iteration also decreases. Though there is a thresh hold up to which this is followed i.e. after certain no of mappers(120) time per iteration starts to rise. This is due to the fact now for simple task more reading has to be done which is overhead.

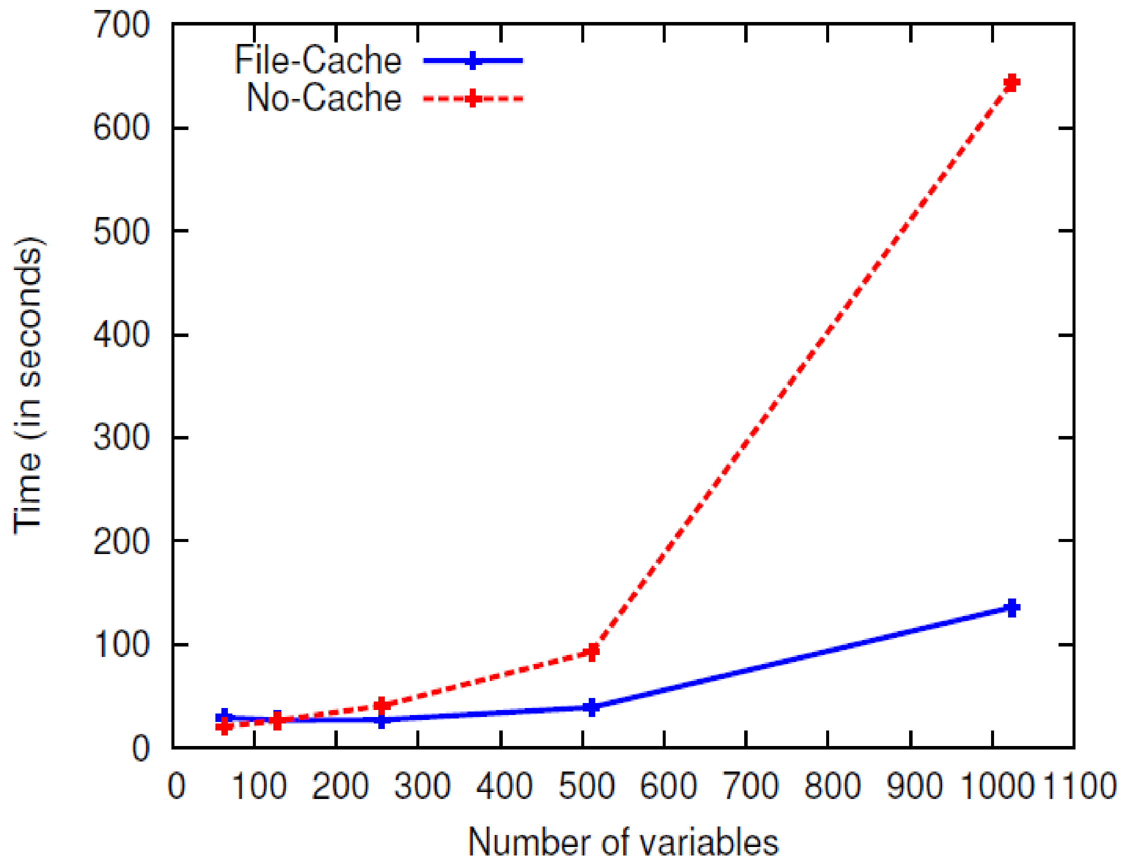


Figure 4.8: Scaling the model building with problem size.

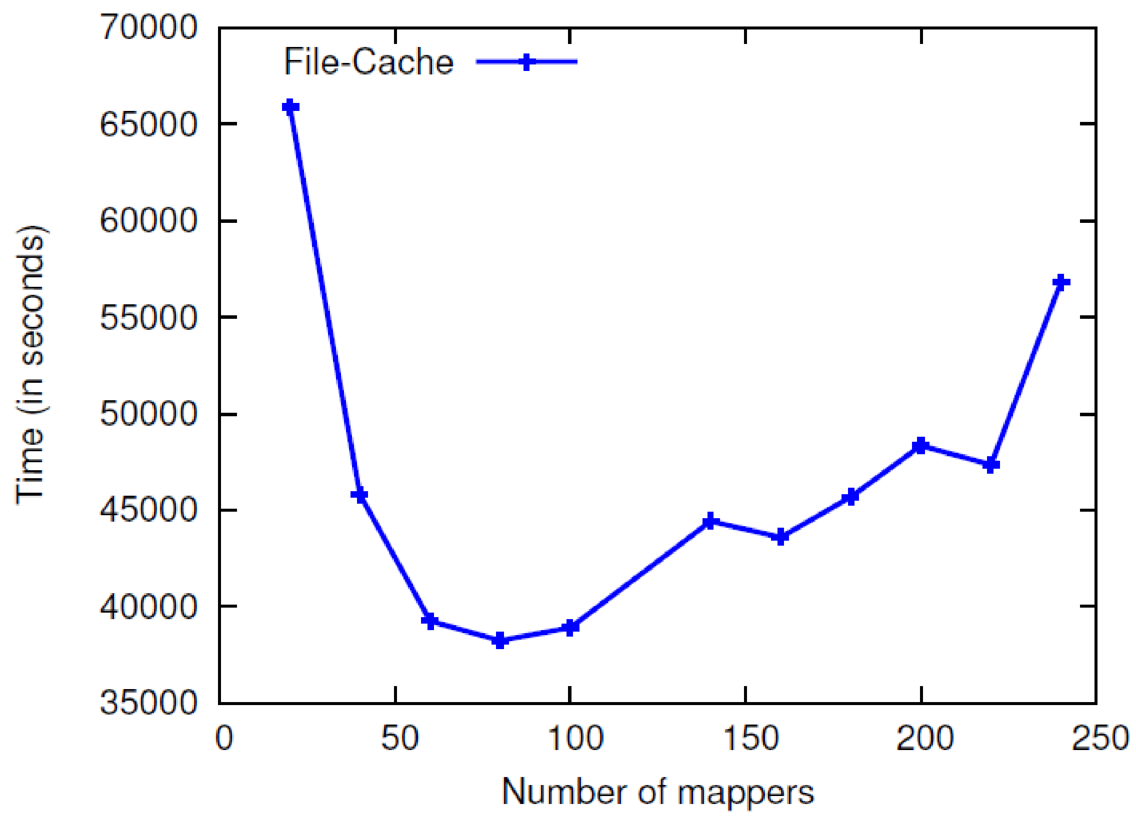


Figure 4.9: Scaling the model building with number of mappers.

Chapter 6
Conclusion

It has been shown that implementation of evolutionary algorithms is possible in parlance to data-intensive computing standard with efficiency. We have shown Step by step transformation of 3 algorithms which based on genetic algorithms in presented .All 3 of the proposed algorithms are compared and analysis is done. It has been shown that evolutionary computational algorithms like GA can me remodeled into Hadoop Map-reduce model and resulting algorithm is easily scalable and hence promote parallelization .in addition to it our results have also shown that when an algorithm which has inherit feature for parallel implementation engineered properly can produce better result than other contemporary ones and can exploit cores in modern multi cored processors better parallel implementation without changing original nature of data flow.

Results have also shown that when we have to deal with the larger problems Hodoop is the correct and optimal choice provided resources are always available, though iteration time should remain same and should not depend on size of problem. Thanks to inherent property of genetic algorithms even linear speedups are possible on data intensive computing.

We can see by our algorithms that not only multi core architectures can be benefitted by this algorithm but also multiprocessor architectures like NUMA .Future work is possible in computation intensive Map phase and random number generation which is possible to be performed in parallel with the Reduce on the Central processing units. Practical implementation of scalable GAs can also be considered for future works.

REFERENCES

[1] D. E. Goldberg, Genetic Algorithms in Search, Optimization and Machine Learning. Addison-Wesley Longman Publishing Co., Inc., 1989.

[2] S. J. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Third. Prentice Hall, 2010.

[3] J. H. Holland, Adaptation in Natural and Artificial Systems. The University of Michigan Press, 1995.

[4] T. White, Hadoop: The Definitive Guide, Third. O'Reiily, 2012.

[5] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters", OSDI, 2004.

[6] C. Jin, C. Vecchiola, and R. Buyya, "Mrpga: an extension of mapreduce for parallelizing genetic algorithms", in eScience, 2008. eScience '08. IEEE Fourth International Conference on, IEEE, 2008, pp. 214–221.

[7] A. Verma, X. Llor`a, D. E. Goldberg, and R. H. Campbell, "Scaling genetic algorithms using mapreduce", in Intelligent Systems Design and Applications, 2009. ISDA'09. Ninth International Conference on, IEEE, 2009, pp. 13–18.

[8] S. Di Martino, F. Ferrucci, V. Maggio, and F. Sarro, "Towards migrating genetic algorithms for test data generation to the cloud", in. IGI Global, 2012, ch. 6, pp. 113–135.

[9] G. Luque and E. Alba, Parallel Genetic Algorithms, Theory and Real World Applications. Springer, 2011.

[10] L. Di Geronimo, F. Ferrucci, A. Murolo, and F. Sarro, "A parallel genetic algorithm based on hadoop mapreduce for the automatic generation of junit test suites", in Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on, IEEE,2012, pp. 785–793.

[11] D. Cutting, Data interoperability with apache avro, Cloudera Blog, 2011. [Online]. Available:http://blog.cloudera.com/blog/2011/07/avro-data-interop.

[12] M. A. Hall, "Correlation-based feature selection for machine learning", PhD thesis, The University of Waikato,1999.

[13] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters", Communications of ACM, 51(1):107–113, 2008.

[14] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler, “Scalable multiobjective optimization test problems,” in Proceedings of Congress on Evolutionary Computation, 2002, pp. 825-830.
[15] S. Di Martino, F. Ferrucci, C. Gravino, V. Maggio, F. Sarro, “Using MapReduce in the Cloud to enhance effectiveness and scalability of genetic algorithms for test data generation,” available at http://www.dmi.unisa.it/people/sarro/www/research/SBST Cloud.html

[16] G. Fraser, A. Arcuri, “Evolutionary generation of whole test suite,” in Proceedings of the 11th International Conference On Quality Software, 2011, pp. 31-40.

[17] D. E. Goldberg, “Genetic Algorithms in Search, Optimization, and Machine Learning,” Addison-Wesley, 1989.

[18] GoogleAppEngine, http://code.google.com/appengine/

[19] CUDA on Hadoop MapReduce, http://wiki.apache.org/hadoop/CUDA%20On%20Hadoop

[20] M. Harman, “The current state and future of search-based software engineering,” in Proceedingd of Future of Software Engineering 2007, pp. 342-357
.
[21] M. Harman, P. McMinn, “A theoretical and empirical study of search based testing: local, global and hybrid search,” IEEE Transactions on Software Engineering, 36(2), 2010, pp. 226-247.

[22] C. Jin, C. Vecchiola, R. Buyya: “MRPGA: An Extension of MapReduce for Parallelizing Genetic Algorithms,” in Proceedings of IEEE Fourth International Conference on eScience, 2008, pp. 214-

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

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

[25] X. Llor`a. Data-intensive computing for competent genetic algorithms: a pilot study using meandre. In GECCO ’09: Proceedings of the 11th Annual conference on Genetic and evolutionary computation, pages 1387–1394, New York, NY, USA, 2009. ACM.

[26] X. Llor`a, B. ´Acs, L. Auvil, B. Capitanu, M. Welge, and D. E. Goldberg. Meandre: Semantic-driven dataintensive flows in the clouds. In Proceedings of the 4th IEEE International Conference on e-Science, pages 238–245. IEEE press, 2008.

[27] X. Llor`a, A. Verma, R. H. Campbell, and D. E. Goldberg. When Huge Is Routine: Scaling Genetic Algorithms and Estimation of Distribution Algorithms via Data-Intensive

Computing. In F. Fernndez de Vega and E. Cant-Paz, editors, Parallel and Distributed Computational Intelligence, chapter 1, page 1141. Springer-Verlag, Berlin Heidelberg, 2010.

[28] T. Maruyama, T. Hirose, and A. Konagaya. 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, 1993. Morgan Kaufmann Publishers Inc.

[29] C. A. Mattmann, D. J. Crichton, N. Medvidovic, and S. Hughes. 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, 2006. ACM.

[30] J. P. Morrison. Flow-Based Programming: A New Approach to Application Development. Van Nostrand Reinhold, 1994.