Chapter 6
Experimental Results

This chapter covers experimental work carried out with the implementation of Hadoop 2.7.2 on Grid'5000. This chapter gives a brief overview of Grid'5000 experiment set-up, followed by parameters used for measurement. Finally, the result is obtained using proposed "Saksham" model and compared with standard Hadoop results.

6.1 Overview of Grid'5000

We have tested our experiment on Grid'5000 (Grid5000.fr, 2018) heterogeneous cluster. Grid'5000 is a large scale distributed testbed for the researchers to experiment with their research on a high configurable cluster. Gri'5000 not only provides storage resources but, it also supports following key features for persistent and reliable infrastructure for researchers.

- **Resource Availability:** Grid'5000 supports a cluster made up of 1000+ nodes. These compute nodes are grouped into homogeneous and heterogeneous clusters. It supports the latest hardware infrastructure such as Xeon Phi- CPUs, latest GPU, SSD drives, 10-25G Ethernet and InfiniBand.

- **Reconfigurable & Controllable:** Grid'5000 provides full access to the researchers for configuration of software and resource usage. It provides a perfect environment for HPC, cloud and Big Data application testing. Due to bare metal deployment features, the researcher can isolate experiments at the network layer.

- **Advanced monitoring and measurement support:** Grid'5000 supports ganglia and other monitoring tools for monitoring of clusters, networks, and other experiments. Grid'5000 also supports the measurement of energy consumption (i.e. kwapi) and analytics of resource usage.

6.2 Experiment Setup

We have used 10 nodes for our experiment (i.e. 1 Namenode and 10 Datanodes). The cluster is configured for Hadoop 2.7.2 version on Ubuntu 14.04 operating system. Appendix II illustrates the Hadoop deployment on Grid'5000. The configuration of nodes is shown in table 6.1. The table displays the heterogeneity of nodes in terms of CPU, memory, storage, number of cores and networks. Table 6.2 shows priority and node label settings for block rearrangement and job processing respectively. In table 6.2 parasilo and parapide are the names of the nodes in the cluster Rennes of Grid'5000.

Nodes are labeled as per the priority that we would like to give to the node. Here 2 means highest priority and 1 means lowest priority. Nodes are labeled or given priority based on the available configuration or available resources on the node. For example, if our job is compute intensive, we will give higher priority and hence we assign label 2 which has more processing capacity. In case if our job is data intensive, then we will give higher priority where more memory and more file space is available, instead of the one with CPU Power.

CPU	Detail Specifications	No of nodes
Intel Xeon E5-2630 v3 **CPU: 2 CPUs/node**	**Cores:** 8 cores/CPU **Memory:** 128 GB memory **Storage:** 558 GB/node, **Network:** 10 Gbps	Parasilo-[1-6] Total – 6
Intel Xeon X5570 **CPU: 2 CPUs/node**	**Cores:** 4 cores/CPU **Memory:** 24 GB memory **Storage:** 465 GB/node, **Network:** 20 Gbps	Parapide-[1-4] Total – 4

Table 6.1 Hadoop 2.7.2 Heterogeneous Cluster Configuration

Priority-2	Priority-1
Node Label – "high_cpu"	Node Label – "low_cpu"
parasilo-1.rennes.grid5000.fr	parapide-1.rennes.grid5000.fr
parasilo-2.rennes.grid5000.fr	parapide-2.rennes.grid5000.fr
parasilo-3.rennes.grid5000.fr	parapide-3.rennes.grid5000.fr
parasilo-4.rennes.grid5000.fr	parapide-4.rennes.grid5000.fr
parasilo-5.rennes.grid5000.fr	
parasilo-6.rennes.grid5000.fr	

Table 6.2 Data Rearrangement Priority and Node Label Settings

6.3 Experiment Scenarios

In this experimental study, we have used three different scenarios for comparing the results. It is important to compare the results of our proposed model with the default and custom configured Hadoop set-up for validating and reliability of proposed results. Figure 6.1 shows the test cycle for all the scenarios discussed below. Results of all scenarios help to compare the results and show the effectiveness of the "Saksham" model.

Scenario 1: In this scenario, we first place data blocks using the default HDFS block placement policy. We use default Hadoop 2.7.2 parameters and schedule the MapReduce jobs using different schedulers for Big Data processing. Results of these applications are evaluated for comparing it with our proposed "Saksham" model.

Scenario 2: In this scenario, the first step remains the same as scenario-1 i.e. placing data blocks using the default HDFS block placement policy. Second, we assigned node labels to the nodes for job placement. Last, we scheduled the MapReduce jobs using various scheduling policies for Big Data processing. We evaluated the result of the node label approach as it is important to show the effectiveness of our proposed approach and to affirm that it is better than mere implementation of default node label settings.

Scenario 3: In this scenario, once again we used the default HDFS block placement policy for data placement. Second, we implemented "Saksham" block rearrangement algorithm which rearranges the block according to the configuration parameter. In this level, all blocks are stored to the nodes which have high computation power or more resource availability (I.e. CPU, memory). Next, we assigned node labels (Refer Appendix III) to the nodes. At last, we scheduled the MapReduce jobs using all scheduling policies.

6.4 Performance Metrics

For our experiment results, we focused on two important parameters which define the performance of Hadoop. We have evaluated and analyzed the result of Hadoop based on data locality, job execution time, and job completion time. These are the

major performance criterions to improve Hadoop performance. In our experiment, block rearrangement time is also important as total time would include the block rearrangement time, to measure the results precisely.

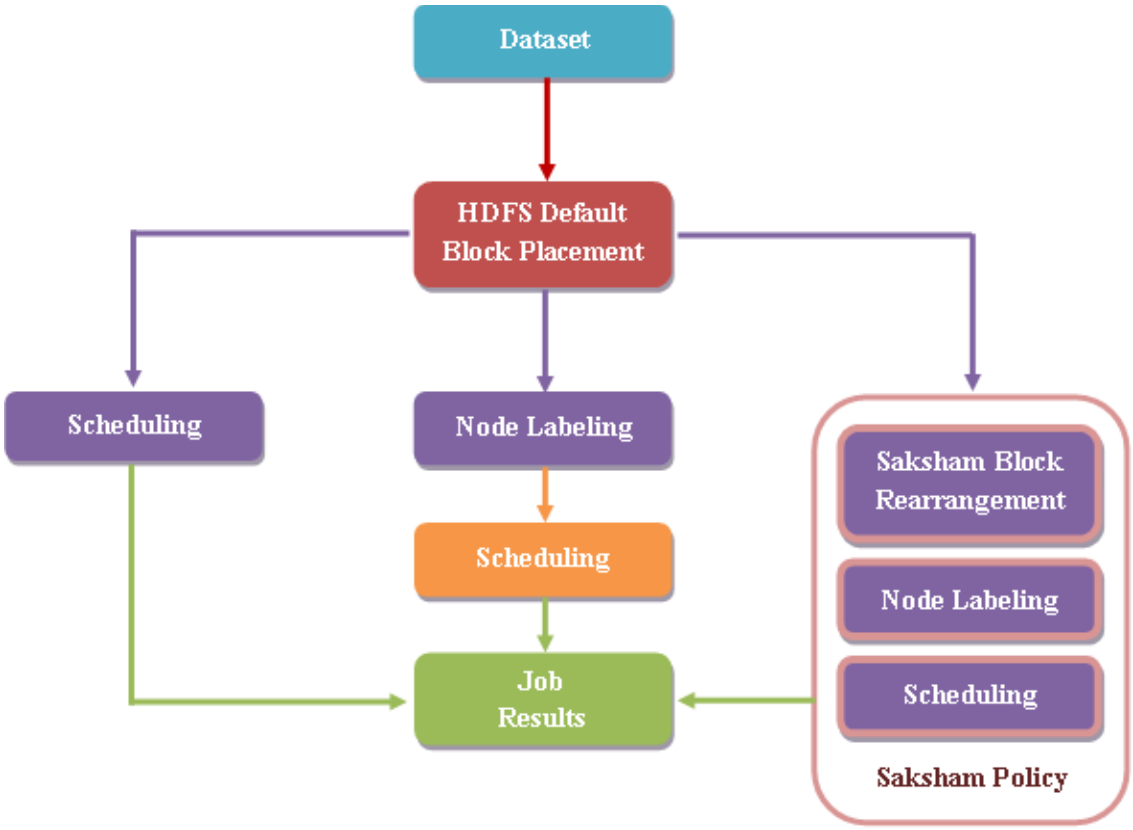


Figure 6.1 Test Cycle for All Scenarios

Data Locality: It refers to process of moving computation close to data rather than transferring large data near the computing resources. If data moves to the cluster, then in some instances may cause network traffic congestion and size of data may skew the overall job performance. Hence, improvement in data locality will definitely enhance overall performance.

For our experiment, we calculated data locality percentage of all applications. Here we have represented, total number of map tasks launched for each application as t_{tsk}, total data local tasks as t_{dl} and finally data locality percentage $locality_{percentage}$ is calculated as,

$$locality_{percentage} = (t_{dl} / t_{tsk}) * 100 \quad \text{Eq.1}$$

Job Execution Time: It is the time required to complete the job once job starts executing. It is obvious that job execution time is a major factor of performance criteria.

For our experiment, we focused on reduction of job execution time so that significant performance can be seen. Here we have represented, job start time as $\boldsymbol{t_s}$, job finish time as $\boldsymbol{t_f}$and finally job execution time $\boldsymbol{t_e}$ is calculated as,

$$\boldsymbol{t_e = t_f - t_s} \qquad \text{Eq.2}$$

Job Completion Time: It is the time required to complete the job once it is submitted by a user. It is important to consider job completion time as many factors play a role once the job is submitted by the user. The job is required to be scheduled, depending on the availability of resources and priority of job.

For our experiment, we strive for reducing the job waiting time so that job completion time can be improvised. Here we have represented, job execution time as $\boldsymbol{t_e}$ according to equation 2, job waiting time as $\boldsymbol{t_w}$ and finally job completion time $\boldsymbol{t_c}$ is calculated as,

$$\boldsymbol{t_c = t_e + t_w} \qquad \text{Eq.3}$$

Rearrangement Time: It is the time required to rearrange the blocks using "Saksham" block rearrangement policy.

In our experiment, our major task is to rearrange the blocks according to the availability of resources. Therefore, it is significant to examine the time taken for the rearrangement of blocks. Here we have represented, block rearrangement start time as $\boldsymbol{t_{sks}}$, rearrangement completion time as $\boldsymbol{t_{skc}}$ and finally rearrangement time $\boldsymbol{t_{skr}}$ is calculated as,

$$\boldsymbol{t_{skr} = t_{skc} - t_{sks}} \qquad \text{Eq.4}$$

6.5 Applications & Dataset

In order to exemplify the legitimacy of our "Saksham" resource aware block rearrangement policy we adopted the widely accepted and benchmark applications

for our experiment. It is equally important to validate the application using proper dataset. Here in next subsection, we have discussed the test applications and dataset used for our experiment.

6.5.1 Test Applications

We ran two benchmark job applications and one Big Data application for testing to prove the efficacy of our proposed approach.

1. **WordCount:** WordCount application counts the total number of times each word occurs in a specified file. The WordCount application is implemented using MapReduce programming to achieve parallelism. Standard "bag of words" (Archive.ics.uci.edu, 2019) static test dataset is used for the counting job. WordCount is typically CPU intensive application as it requires to compute the total words after each map-reduce phase.
2. **TeraSort:** TeraSort sorts the large data generated by TeraGen using MapReduce programming. The TeraSort benchmark is the most well-known Hadoop benchmark (Ibm.com, 2019b) to sort the data. TeraSort is typically I/O intensive application as it requires to read/write large data between CPU and memory.

3. **DNA Pattern Search:** In DNA pattern search, it searches for an exact subsequence in a given DNA sequence. This application is vital in many areas such as genetics, genomics, forensics, pharmacogenetics, phylogeny etc. However, it is very much time consuming due to its large datasets (i.e. Big Data) and pattern search processing. We opt for this application to prove that our "Saksham" approach is not only efficient for benchmark applications but can also be useful for real-world applications.

6.5.2 Dataset Description

For our experiment, we used datasets of size 10 GB and 20 GB. For WordCount application we use text data collection (Archive.ics.uci.edu, 2019) of 10 GB and 20 GB size. Dataset contains different words in vocabulary. For TeraSort we use the data generated using TeraGen utility. TeraGen generates and writes the large

dataset on to the cluster nodes. For DNA pattern match we used standard human genome dataset from NCBI (ncbi.nlm.nih.gov, 2019a) website. Dataset is of size 10 GB which includes genomes of Homo sapiens.

6.6 Result Analysis

In this section, we analysed the results of three applications. The performance of "Saksham" model is explored and is compared with the results of default Hadoop setup along with node label settings.

6.6.1 Benchmark Applications

In our experiment, two Hadoop applications WordCount and TeraSort were implemented for testing. WordCount and TeraSort applications are archetypal MapReduce jobs for Hadoop performance measurement. We used 4 datasets as the input data out of which 2 text dataset files (5GB and 10 GB) for WordCount and 2 datasets (5GB and 10 GB) were generated using TeraGen utility. For this experiment we kept replication factor as 3. Therefore total block size was 15 GB and 30 GB.

As described in experiment scenarios; *we* first loaded the dataset on HDFS cluster using default HDFS block placement policy. Afterwards, we applied "Saksham" block rearrangement policy according to settings described in table 6.2. Figure 6.2 [A-B] shows the result of our approach on 15 GB and 30 GB text dataset we used for WordCount application. As it is clearly shows that default HDFS block placement policy places data blocks unevenly among the cluster nodes. Moreover, processing capability of nodes was not considered while placing the blocks. Results also shows that after applying the "Saksham" block rearrangement policy blocks were rearranged according to the priority set in configuration.xml, as shown in table 6.2. We assigned priority-2 for the nodes parasilo-[1-6] as these nodes have more resources (number of cores, storage and memory) compared to the nodes parapide-[1-4] shown in table 6.1.

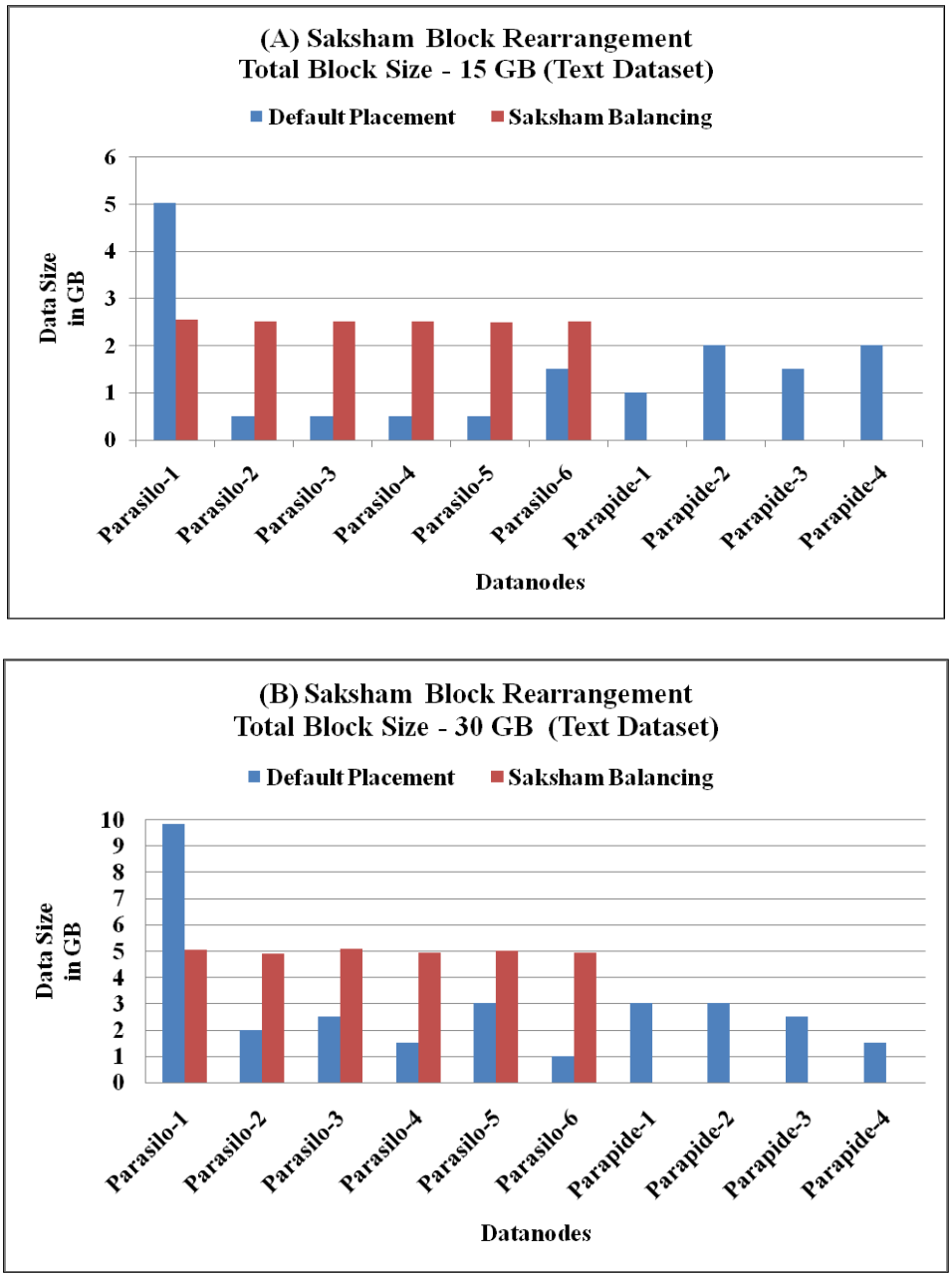


Figure 6.2 Saksham Balancing: Text Dataset (A) Size-15 GB (B) Size- 30 GB

Figure 6.3 [A-B] depicts the result of our approach on 15 GB and 30 GB dataset we use for TeraSort application. Here also we first used default HDFS block placement policy for data placement and then we applied "Saksham" block rearrangement policy for rearrangement.

Result of fig. 6.2 and 6.3 clearly illustrates that our proposed "Saksham" block rearrangement successfully rearranges the blocks to the nodes which have priority-2. It is very much important to rearrange the blocks to the only nodes which have more processing capability which we successfully achieved.

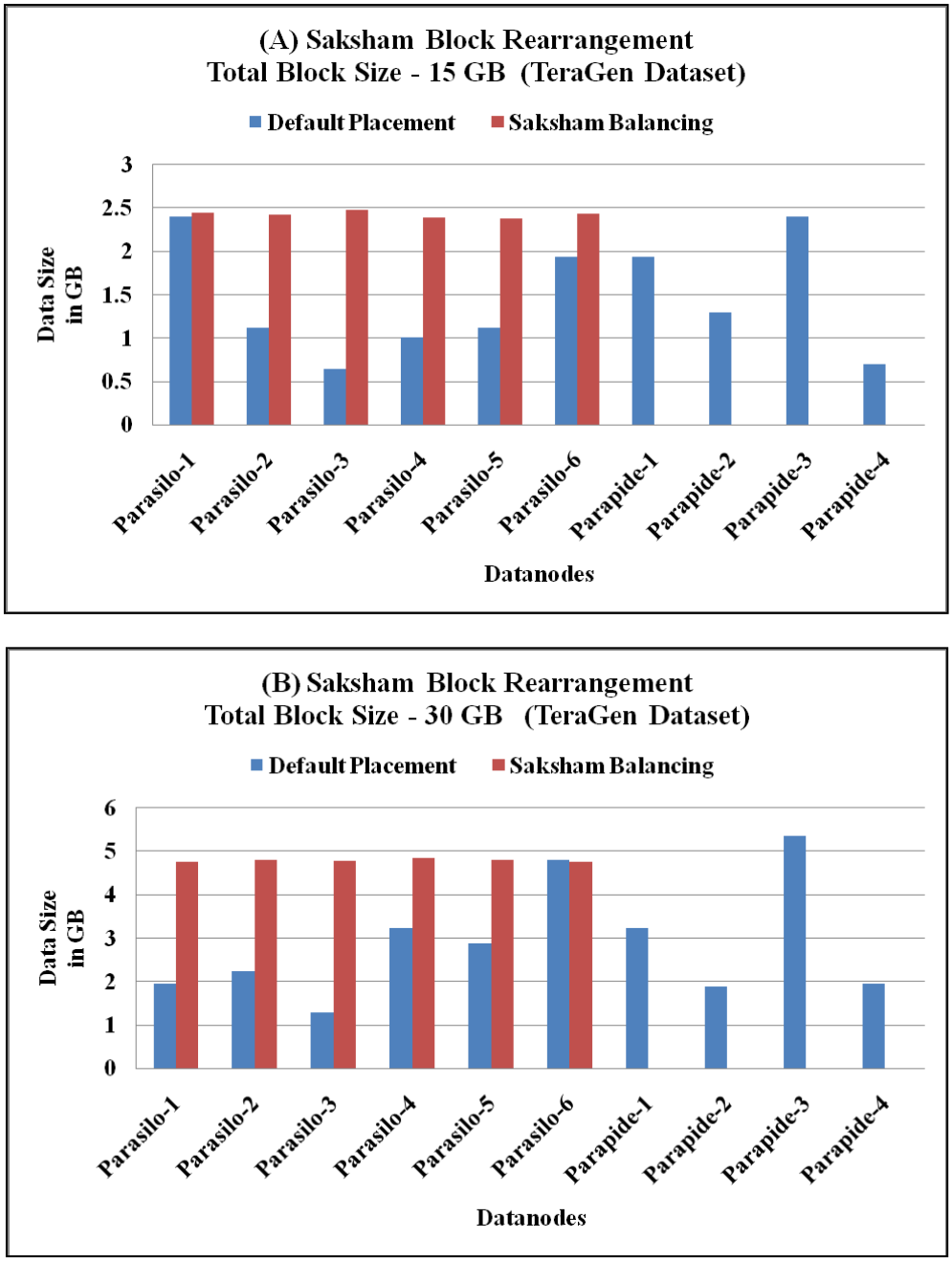


Figure 6.3 Saksham Balancing: TeraGen Dataset (A) Size- 15 GB (B) Size-30 GB

Table 6.3 shows storage utilization after implementing default block placement policy and in the next step after applying "Saksham" block rearrangement policy. The result of table 6.3 proves that our "Saksham" algorithm is successfully configured and all the blocks were rearranged to the nodes which have prirority-2 and disk utilization of nodes was also merely same. Figure 6.4 shows rearrangement time taken by the "Saksham" algorithm.

	Text Data 15 GB		Text Data 30 GB		TeraGen 15 GB		TeraGen 30 GB	
Datanodes	**Default Placement**	**Saksham Balancing**	**Default Placement**	**Saksham Balancing**	**Default Placement**	**Saksham Balancing**	**Default Placement**	**Saksham Balancing**
Parasilo-1	5.04	2.55	9.85	5.08	2.41	2.45	1.94	4.77
Parasilo-2	0.50	2.53	2.02	4.93	1.12	2.43	2.23	4.80
Parasilo-3	0.50	2.51	2.52	5.09	0.65	2.49	1.29	4.79
Parasilo-4	0.50	2.51	1.51	4.95	1.01	2.39	3.23	4.85
Parasilo-5	0.50	2.51	3.02	5.01	1.12	2.38	2.87	4.80
Parasilo-6	1.51	2.52	1.01	4.96	1.94	2.44	4.81	4.76
Parapide-1	1.01	28 kb	3.03	28 kb	1.94	28 kb	3.23	28 kb
Parapide-2	2.02	28 kb	3.02	28 kb	1.29	28 kb	1.87	28 kb
Parapide-3	1.51	28 kb	2.52	28 kb	2.41	28 kb	5.35	28 kb
Parapide-4	2.02	28 kb	1.51	28 kb	0.70	28 kb	1.94	28 kb

Table 6.3 Disk Utilization of All Nodes for Different Data size

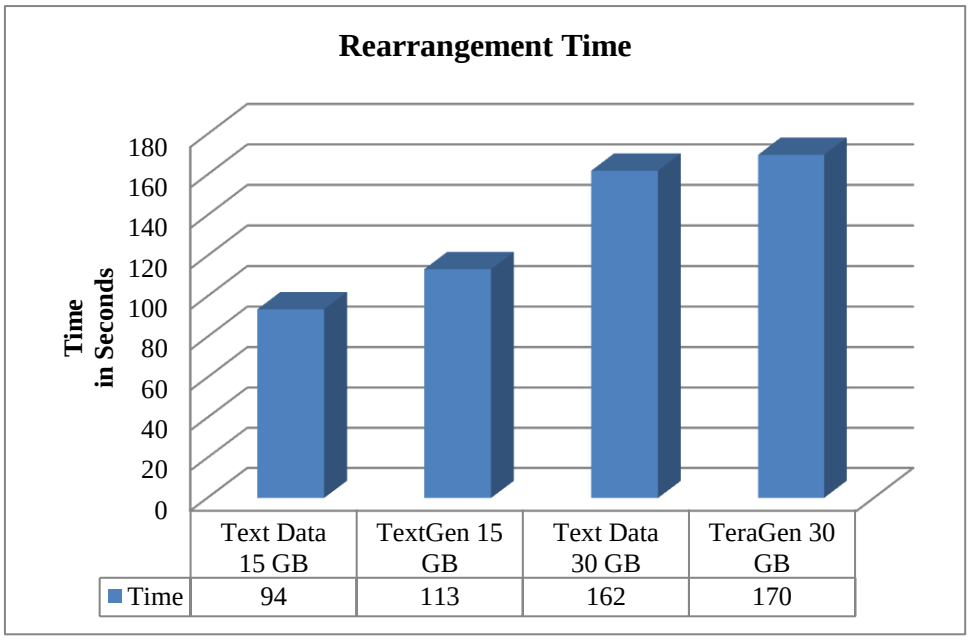


Figure 6.4 Saksham Block Rearrangement Time

We have successfully achieved control over block rearrangement based upon the priority assigned to the nodes and also all nodes have approximately equal data size. Therefore, along with rearranging the block we also resolved load balancing issue. Next, we assigned Node Labels to the nodes according to the table 6.2 and using YARN resource manager we scheduled the jobs using capacity (Issues.apache.org, 2019a) and fair (Issues.apache.org, 2019b) schedulers according to given labels. Then we compared our proposed approach with the all 4 scheduling policies described in chapter 3. We evaluated and compared the performance of our

proposed "Saksham" policy in accordance with node locality percentage, job completion time, job execution time and total time.

We used two datasets of size 10 GB and 20 GB for our experiment. We executed these experiments 15 times on Grid 5000 and the values of time or data blocks in GB is the average time of these 15 executions. We focused primarily on one of the major parameters of performance improvement in Hadoop, which was data locality. If data locality improves proportionally, it improves MapReduce job processing time. For Hadoop performance data locality is an important measure as in case of Hadoop, computation is moved where data is residing. But if data is residing at random nodes, where containers/processing resources are not available or slow in nature, then Hadoop by default will move the data to another node in the same rack or across node on another rack, which will consume data relocation time and also network bandwidth, resulting into higher job execution time and more amount of resources. So, to avoid moving data at runtime, Saksham policy places blocks only on those nodes where computational and sufficient storage resources are already available.

We compared our strategy with default placement execution and after applying only node labelling without "Saksham" balancing. Table 6.4 illustrates the result of total tasks launched, data local found and data locality in percentage. Figure 6.5 shows the comparative result of data locality achieved by various strategies. In fig 6.5 each bar represents the data locality achieved by default Hadoop, after applying Node label, and proposed "Saksham" approach. Results validate that our "Saksham" algorithm with node labelling approach accomplishes almost 90% data locality which was far better than other strategies.

		Default			Node Label			Saksham		
Jobs	**Data Size**	**Total Task Launched**	**Data Local**	**Data Locality %**	**Total Task Launched**	**Data Local**	**Data Locality %**	**Total Task Launched**	**Data Local**	**Data Locality %**
Word Count	**10 GB**	128	88	68.75%	122	74	60.66%	120	112	93.33%
	20 GB	240	178	74.17%	227	140	61.67%	233	208	89.27%
TeraSort	**10 GB**	134	102	76.12%	126	89	70.63%	127	111	87.40%
	20 GB	267	183	68.54%	250	186	74.40%	236	213	90.25%

Table 6.4 Data Locality Results for Various Strategies

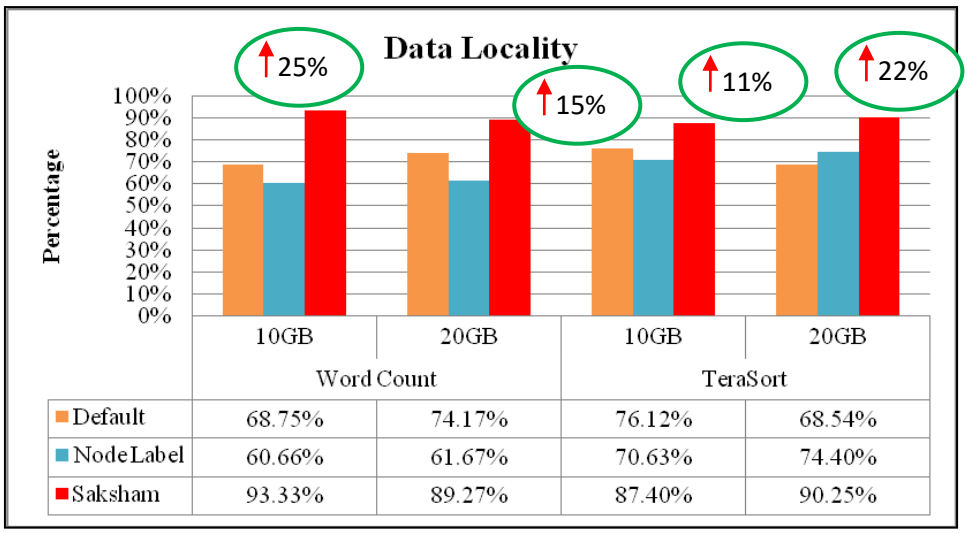


Figure 6.5 Comparison of Data Locality Achieved

Last, we used default Hadoop schedulers for our test. We combined our "Saksham" algorithm plus node labelling and scheduled the jobs for testing. We tested using following schedulers to see the effectiveness: capacity and fair scheduler. Fair scheduler has three policies: Fair-FIFO, Fair-Fair and Fair-DRF. Test scenarios for all experiments discussed in previous section and fig. 6.1 displays our test cycle that we implemented for comparison. We compared the job execution time of our proposed approach with default MapReduce execution, executing using node labelling w/o "Saksham" balancing.

Figure 6.6-6.9 demonstrates the results of job execution of two jobs using two different size of datasets. It is important to note that "Saksham" policy results are with default Hadoop, and after applying node label.

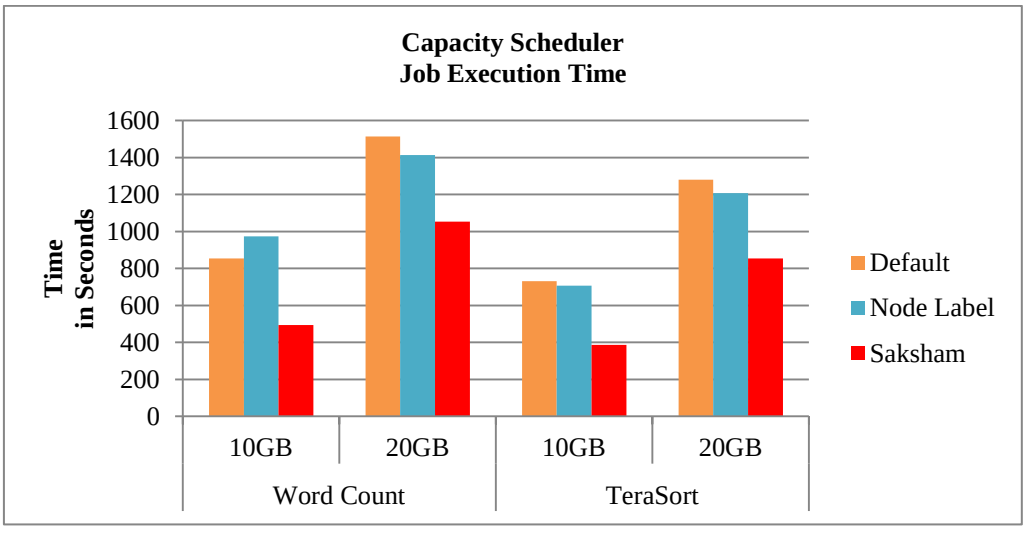


Figure 6.6 Job Execution Time using Capacity Scheduler

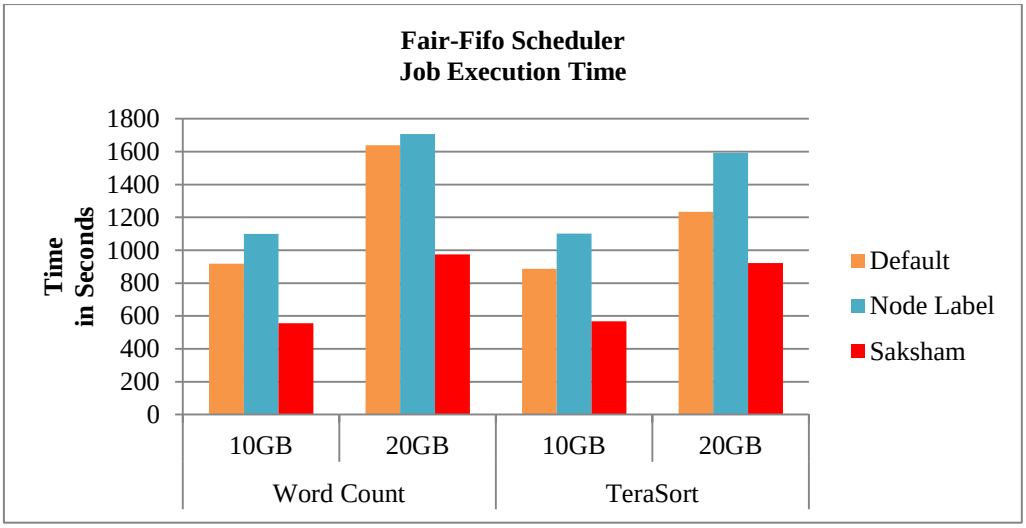


Figure 6.7 Job Execution Time using Fair-FIFO Scheduler

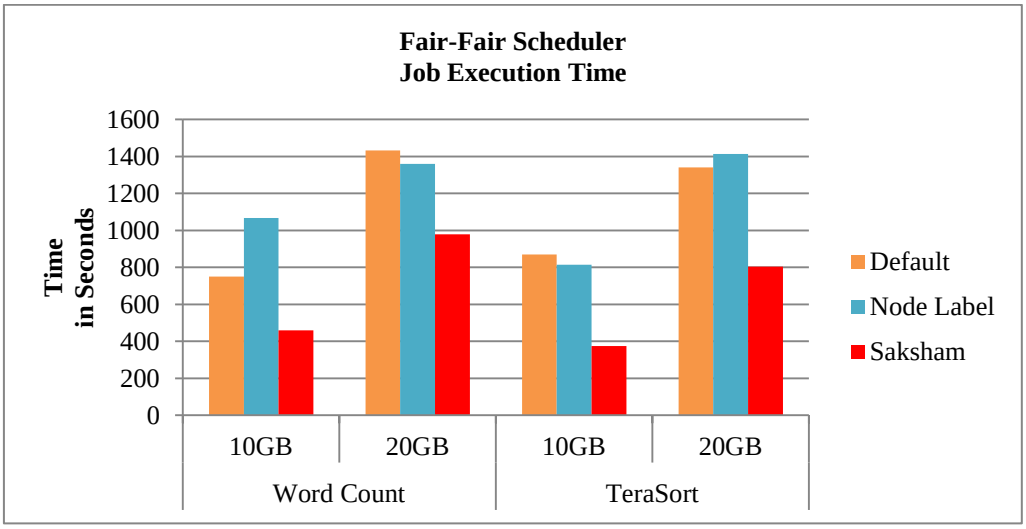


Figure 6.8 Job Execution Time using Fair-Fair Scheduler

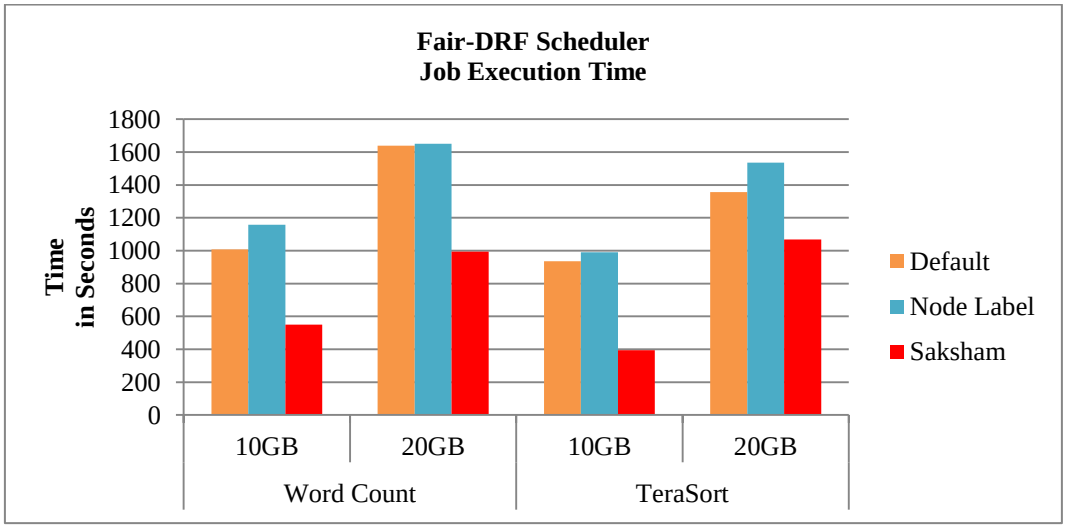
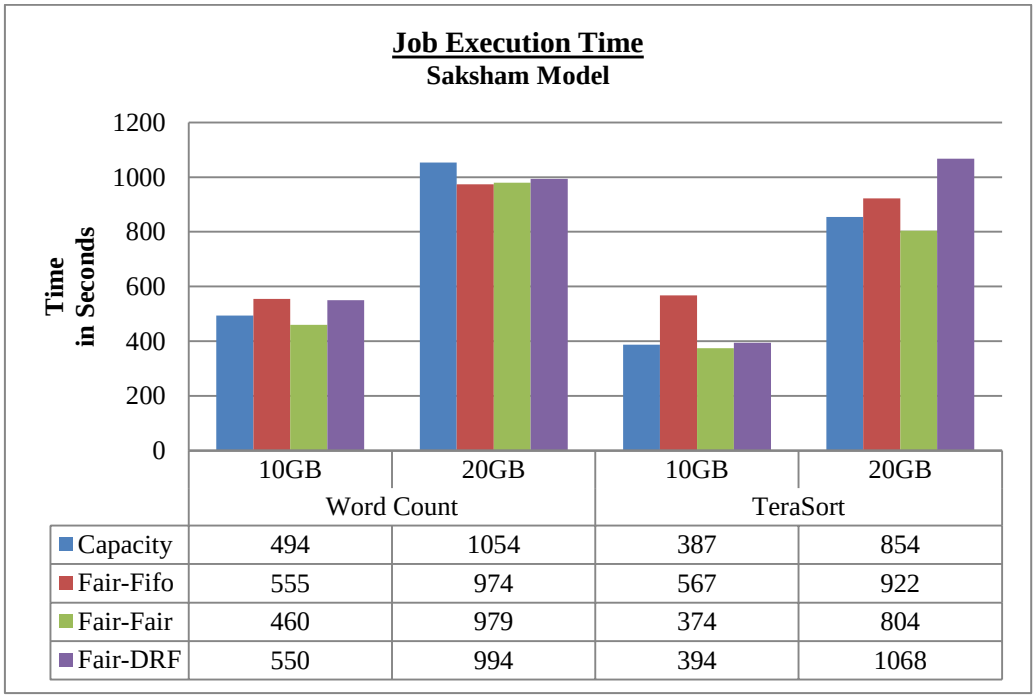


Figure 6.9 Job Execution Time using Fair-DRF Scheduler

Results establish that mere implementation of node labelling creates overhead of the internode and interrack block transfer and augments the job execution time. But our "Saksham" algorithm in combination with node labelling achieves optimized result. Figure 6.10 shows the comparison of the job execution time of different scheduling policies for "Saksham" model. Results of the two applications are compared with two different dataset size. Here fig. 6.10 affirms that fair-fair scheduling strategy outsmarts capacity, fair-fifo, and fair-drf policies.



	Word Count 10GB	Word Count 20GB	TeraSort 10GB	TeraSort 20GB
Capacity	494	1054	387	854
Fair-Fifo	555	974	567	922
Fair-Fair	460	979	374	804
Fair-DRF	550	994	394	1068

Figure 6.10 Job Execution Time of all Scheduling Policies

Results testify that "Saksham" policy successfully attains better job execution time and improves the Hadoop performance. But at the same time, it is important to prepend the amount of time required for the rearrangement of the blocks. Figure 6.11 shows the total time comparison in fair-fair scheduling policy. Here total time refers to job execution time plus block rearrangement time. It is significant to note that even if we consider rearrangement time we achieved substantially surpassing results than the existing policies.

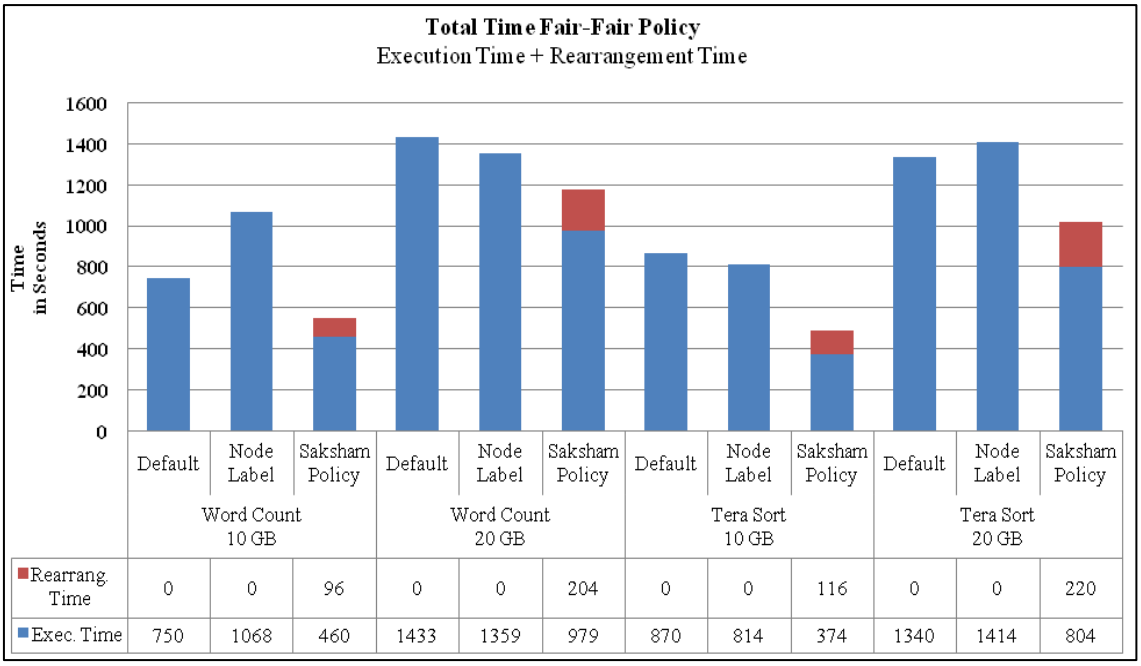


Figure 6.11 Total time (Execution Time + Rearrangement Time) using Fair-Fair Policy

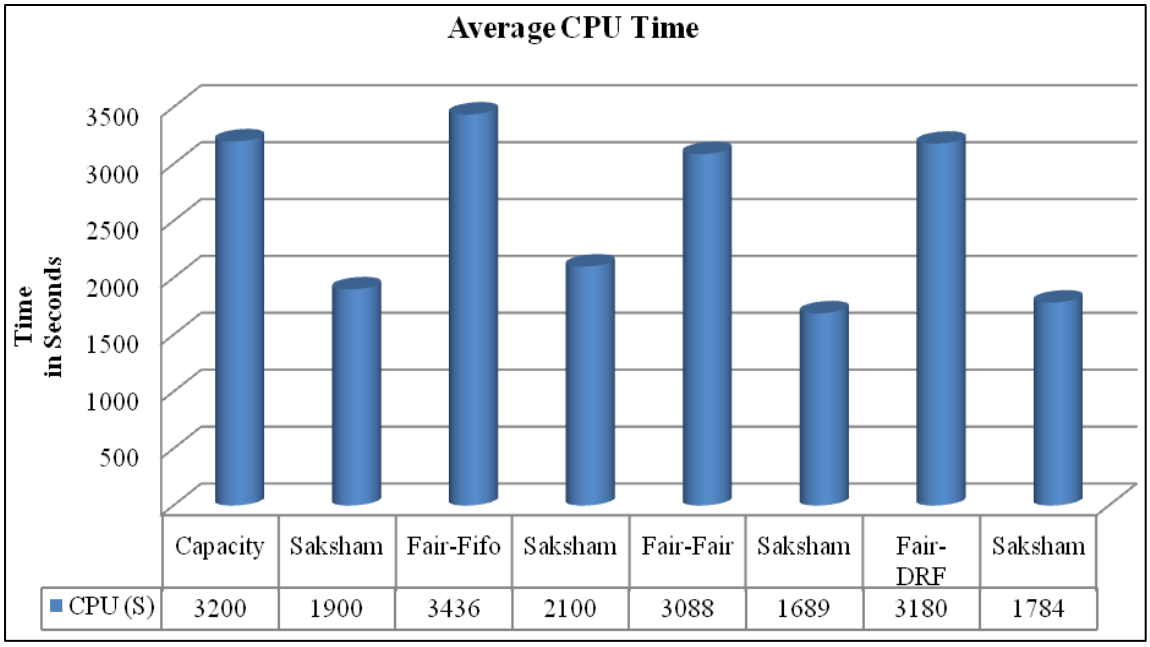


Figure 6.12 Average CPU Utilization

Figure 6.12 and 6.13 shows the comparison of average CPU time and memory utilization for both the jobs. The figure clearly shows that CPU time is less for "Saksham" model compared to all four scheduling policies. In terms of CPU time & memory usage shown in fig. 6.12 and 6.13, Fair-Fair is better as it utilizes less CPU time and effectively uses physical memory. Results show that our "Saksham" approach utilized more memory compared to default Hadoop, but total CPU time is

reduced significantly. Therefore, we can say we reduce the time complexity of resource utilization which ultimately succeeds to speed up our Hadoop performance.

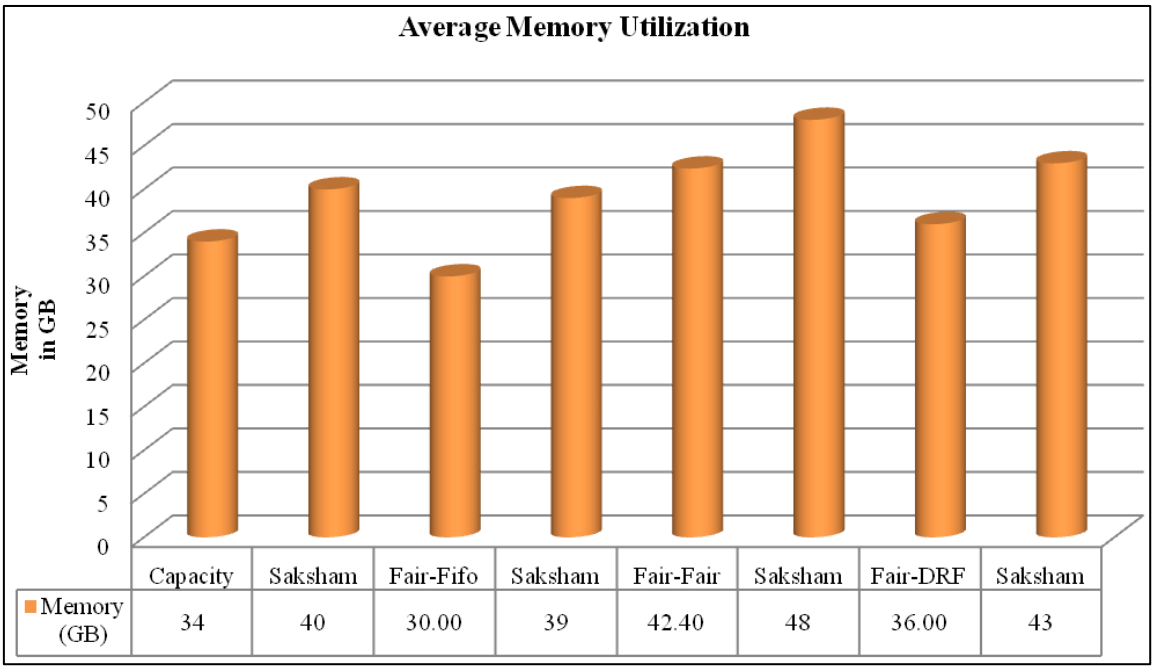


Figure 6.13 Average Memory Utilization

6.6.2 DNA Pattern Search Application

We have also implemented the proposed approach of Saksham Block Rearrangement policy and node labelling for custom application, other than the Benchmark Application of WordCount and TeraSort, to prove the applicability of our proposed algorithm. We implemented the proposed algorithm for DNA Pattern Search Application.

The importance of DNA pattern search is increasing exponentially in Life Sciences research. DNA pattern search plays a vital role in genomic data analysis which has become the need of the day for betterment of mankind. Human DNA data is a massive collection of large sequences. Variety of tools are available to analyze and search patterns in DNA sequences. However, most of these tools need further improvisation (Vineetha, Biji and Nair, 2019) due to the huge size of DNA data.

For our experiment purpose we have used Human Papillomavirus Type 16 (HPV16) (data source: ncbi.nlm.nih.gov, 2019b) for searching subsequence is available in dataset or not. We have used 2 different types of data sets both from ncbi, where HPV16 is the pattern that is searched in the Genomic sequence of Homo sapiens genomic sequence. Human Papillomavirus (HPV16) is usually found in a person having cervical cancer. Cervical cancer is a serious health problem occurs in women, is the third leading malignancy among women after breast and colorectal cancer (Denny et al., 2015). Most types of cervical cancer are caused by human papillomavirus (HPV), a sexually transmitted infection.

We used different configuration of node for DNA pattern search application just to check the authenticity of work and check impact on results. The configuration of nodes is shown in table 6.5. Table 6.6 shows priority and node label settings for block rearrangement and job processing respectively.

CPU	Detail Specifications	No of nodes
Intel Xeon E5-2630 CPU: 2 CPUs/node	**Cores:** 6 cores/CPU **Memory:** 32 GB memory **Storage:** 598 GB/node, **Network:** 10 Gbps	taurus-[1-6] Total – 6
AMD Opteron 250 CPU: 2 CPUs/node	**Cores:** 1 core/CPU **Memory:** 2 GB memory **Storage:** 73 GB/node, **Network:** 20 Gbps	sagittaire-[32-34,44] Total – 4

Table 6.5 Heterogeneous Cluster Configuration (DNA Pattern Search)

Priority-2	Priority-1
Node Label – "high_cpu"	Node Label – "low_cpu"
taurus-1.lyon.grid5000.fr	sagittaire-32.lyon.grid5000.fr
taurus-2.lyon.grid5000.fr	sagittaire-33.lyon.grid5000.fr
taurus-3.lyon.grid5000.fr	sagittaire-34.lyon.grid5000.fr
taurus-4.lyon.grid5000.fr	sagittaire-44.lyon.grid5000.fr
taurus-5.lyon.grid5000.fr	
taurus-6.lyon.grid5000.fr	

Table 6.6 Data Rearrangement Priority and Node Label Settings (DNA Pattern Search)

We used a dataset of size 10 GB for our experiment and results are prepared by taking an average of 15 experiments. We compared our strategy with default placement execution and after applying only node labelling without "Saksham"

balancing. Figure 6.14 illustrates the comparative result of data locality achieved by various strategies. Results confirm that our "Saksham" algorithm with node labelling approach accomplishes almost 93% data locality which was far better than other strategies.

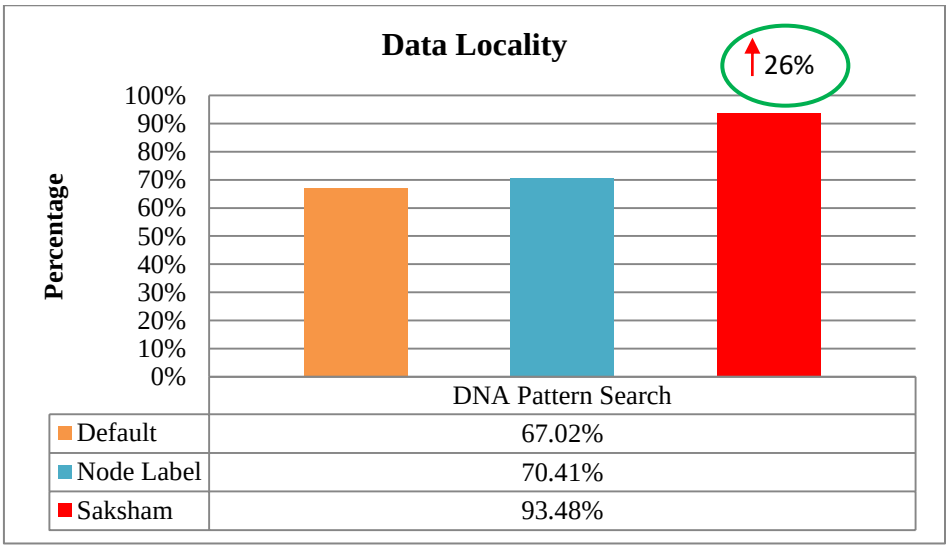


Figure 6.14 Data Locality for DNA Pattern Search Dataset

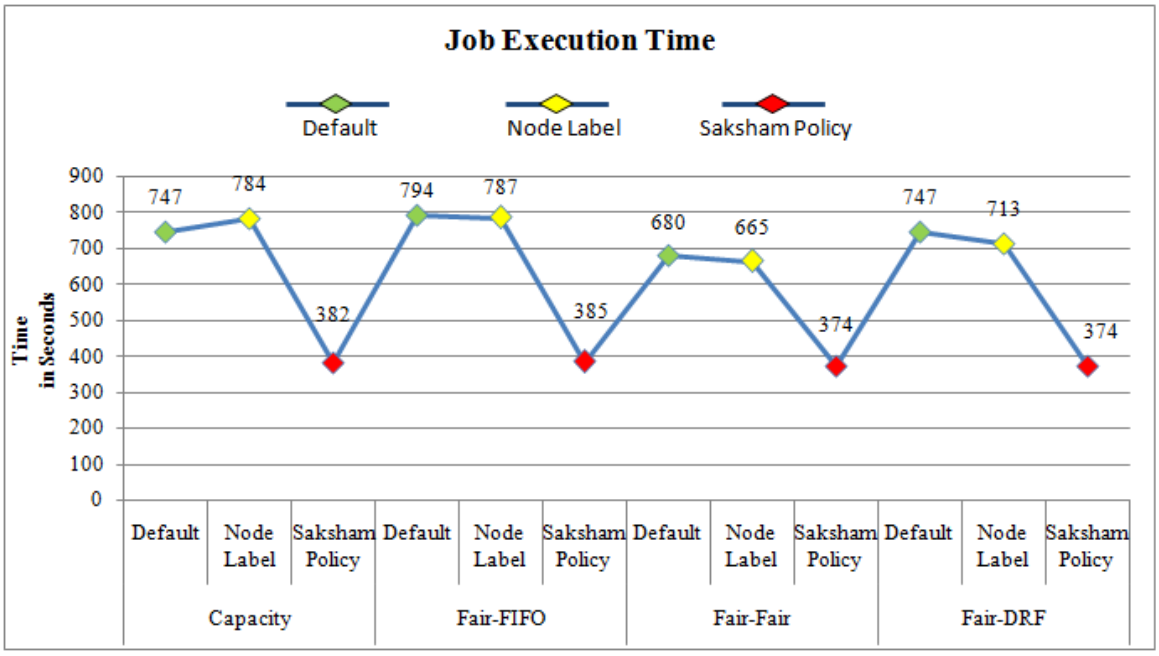


Figure 6.15 Job Execution Time for DNA Pattern Search

Results as shown in fig 6.15 prove that "Saksham" policy successfully accomplishes better job execution time for DNA pattern search application. Figure 6.16 shows the total time comparison of all scheduling policies. It is significant to

note that even if we consider rearrangement time we achieved substantially improvement, surpassing the results of existing policies.

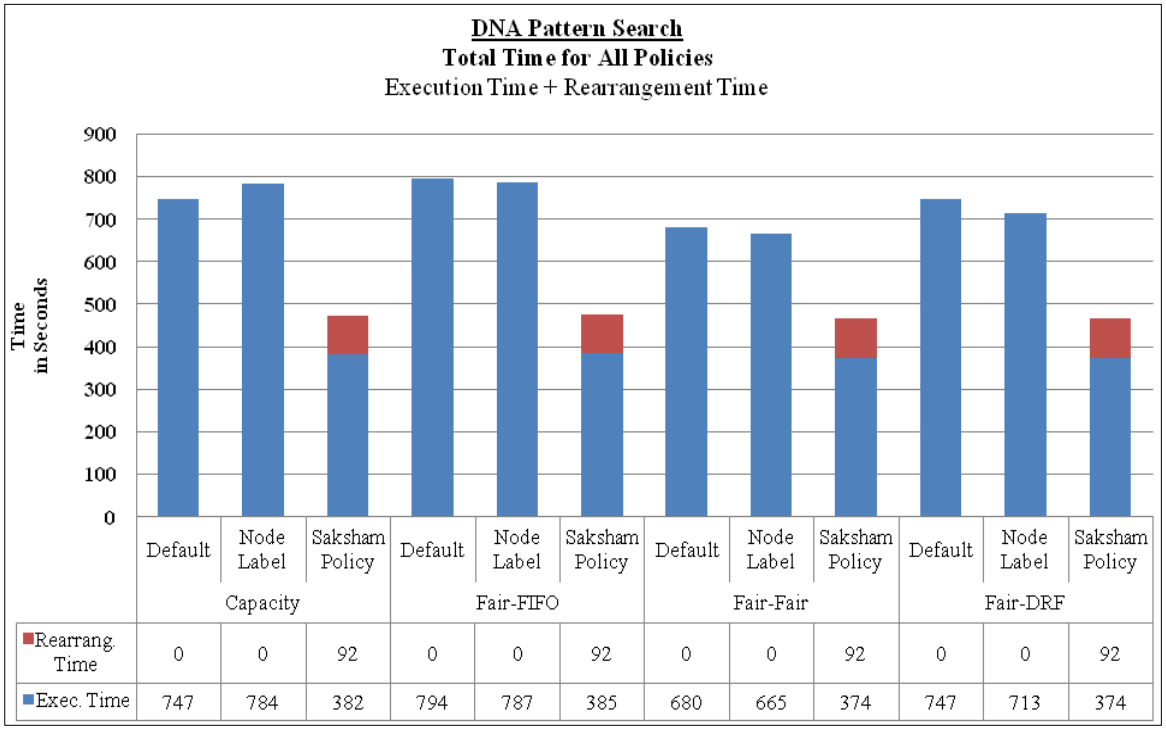


Figure 6.16 Total time (Execution Time + Rearrangement Time) for All Policies

In this chapter, we implemented and tested our proposed "Saksham" model for Hadoop performance improvisation. All three application results are analyzed based on execution time, data locality, and CPU/memory utilization by each application. We studied the performance of the Hadoop in detail and developed a comprehensive "Saksham" model for Big Data processing on a heterogeneous cluster environment. The results of "Saksham' model validate our model as we received more than 90% data locality and execution time is also reduced to approximately 50% than the default execution environment. After the implementation and testing of results, we firmly believe that "Saksham" model is pretty much useful for both homogeneous and heterogeneous environment and also optimizes the performance of Hadoop.