Neural, Parallel, and Scientific Computations 17 (2009) 31-46

A LOAD BALANCING FAULT-TOLERANT ALGORITHM FOR HETEROGENEOUS CLUSTER ENVIRONMENTS

E. M. Karanikolaou and M. P. Bekakos
Laboratory of Digital Systems, Department of Electrical and Computer Engineering, Democritus University of Thrace, 67100, Xanthi, Hellas

Abstract
Herein, a fault-tolerant parallel pattern matching algorithm with load balancing support is presented, targeting at both homogeneous and heterogeneous clusters of workstations from the perspective of computational power. The algorithm is capable of handling up to *(n-1)* faults, introduced at any time, with n being the total number of cluster nodes. It is capable of handling either permanent faults or transient failure situations, temporarily handled as permanent, due to network delay, and thus, nodes may be returned at any time. The experimental results exhibit that the algorithm is capable of returning reliable results in acceptable time limits.
Keywords - Parallel pattern matching, application level fault-tolerance, heterogeneous cluster

1. INTRODUCTION

Clusters of workstations are becoming more and more the current trend for distributed scientific computing, as they are a cost effective solution compared to high-end parallel supercomputers. However, the drawback of this trend is the increasing probability for node or link failures, as the number of nodes increases, due to the relatively cheap components that clusters consist of. If a single failure occurs, it can halt the tasks execution and tasks have to be re-executed, losing all computation carried out so far; hence, hardware failures must be tolerated to save all computation carried out so far. In order to increase the reliability of the system, fault tolerance is indispensable. Moreover, due to the rapid advance in performance commodity computers, when such clusters are upgraded by adding new nodes or even by substituting the failed ones, they become heterogeneous; thus, necessitating the implementation of applications that take into account the different characteristics of the cluster components, in order to achieve optimum results as far as execution time is concerned.

The most natural way to achieve reliable results, in case of failures, is by saving information concerning the state and progress of an application during the computation phase. When components fail, the application can be returned or rolled back to its last saved state and computation can proceed from that stage onwards. Checkpointing and rollback recovery are very useful techniques to implement fault-tolerant applications. There are two general approaches for checkpointing, one in the system level and the other

Received December 17, 2008

in the application level. System level checkpointing, in parallel and distributed computing settings, issuing coordinating checkpoints to define a consistent recovery line, has received the attention of various researchers, as summarized in (Elnozahy et al., 2002). This approach integrated in the message passing library, requires no action from the programmer, but induces a high overhead and is generally difficult to handle in a heterogeneous environment. On the contrary, application level fault-tolerance is a low cost fault-tolerance scheme for detecting and recovering permanent or transient failures, it is application specific and the main considered cost is the effort of the programmer to incorporate this scheme into an application. Manually inserting code to save and recover an application state is a very error prone process (Camargo et al., 2004).

This work concentrates on exact string matching, a subcategory of the pattern matching problem, whose purpose is to find all strict occurrences of a pattern string in a large text file [Charras & Thierry, 2004]. Using application level checkpointing, fault-tolerance support has been added on a relaxed parallel implementation of the exact brute force pattern matching algorithm. Moreover, a load balancing algorithm is also applied in order for the application to scale up efficiently in heterogeneous cluster environments. The experimental results show that the application can handle various failure situations and scales efficiently as the number of processors increases.

The structure of the paper is organized as follows. Section 2, begins with the description of the relaxed parallel brute force pattern matching algorithm. Section 3, describes the steps that are required to incorporate fault-tolerance support into the parallel pattern matching application, described in Section 2, and the issues that one must keep in mind in order to implement similar applications. Section 4, describes the load balancing algorithm used for application's execution in heterogeneous workstations of clusters. The experimental results for the proposed algorithms are presented in Section 5. Finally, Section 6 concludes the investigation.

2. PARALLEL BRUTE FORCE PATTERN MATCHING ALGORITHM

The implementation of the parallel pattern matching application is based on the sequential version of the brute force algorithm and is relaxed; namely, no communication is needed between nodes during the searching phase. The brute force algorithm is not an optimal pattern matching algorithm, but it has been chosen, due to its simplicity and its computing power demands, since the purpose of this investigation is not to compare parallel pattern matching approaches, but to demonstrate the methodology used to incorporate fault-tolerance and load balancing support in such an application as well as to evaluate the performance of the application in a heterogeneous cluster environment.

Before executing the application, a search file of usually huge size is transferred to all participating nodes. The search file being local to all nodes leads to a lower execution time, as no network traffic is necessary during runtime. The same approach may be adopted for the pattern file without having any substantial effect on the results, due to the small size of the pattern, as shown from the experimental results. Afterwards, each node is set to execute the same instructions over a different data stream, according to the identification number assigned by MPI, corresponding with the SPMD parallel

programming model. After having all nodes finished the searching phase, i.e., the part of the search file corresponding to each one of them, the results of occurrences are accumulated to one node using an MPI reduction operation.

Workload for the nodes is based on the total number of the cluster nodes. The search file size is divided with the number of nodes and each node is assigned with the same amount of work, ignoring specific node characteristics. This partitioning gives good performance gains in homogeneous clusters. However, in heterogeneous clusters the performance is decreased since the total execution time depends on the processing capacity of the slowest nodes. A load balancing algorithm for heterogeneous clusters is presented in Section 4.

3. FAULT-TOLERANCE ALGORITHM

In case one or more of the cluster nodes fail, the previous application has to be re-executed from the beginning, losing all the results obtained up to then. In order to handle this situation, checkpointing and rollback recovery are necessary. So, to do this, the master-slave programming paradigm has been used. Fault-tolerance algorithm includes checkpoint critical values during runtime, failure detection and recovery of the failed nodes. Master node is the one who keeps all the critical values for the workers' computational progress state and is responsible for the detection of possible failures and the reassigning of the remaining work to the remaining workers. In order for the application to be able to handle upon *(n-1)* fails, where *n* is the total number of nodes, master is always arranged to recover the first failed node.

The critical information that has to be checkpointed at regular intervals for the pattern matching application is the file pointer, indicating the work progress and the number of occurrences found till then, for each worker node. Master node collects this information and stores it in a checkpoint table, which will be used later on either for fault detection and recovering or, in case of no occurred faults, for occurrences summation and application termination. Every time a node finishes its work, it is marked in a finish table, also created in master. The communication between the master and the worker nodes is performed through message passing with appropriate MPI routines. This communication takes place using threads that interact with the main thread of the program for each node. Thus, the overhead of the fault-tolerance mechanism is small and minimally affects the total execution time of the application (Efremides & Ivanov, 2006).

Detection of failures is done by checking the finish table after all communication has stopped. At this point it is essential to refer to the kind of failures that may occur. There are failure situations in which the master node permanently stops receiving messages from one or more nodes - *category of permanent faults* - and failure situations where master node stops receiving messages for a period of time due to network delay, but eventually messages arrive - *category of transient faults*. The application is capable of handling both types of faults, behaving accordingly for each one of them.

In case permanent faults have occurred, the recovery of the detected as failed nodes, takes place, retrieving information from the checkpoint table about their last saved progress and assigning their remaining work to the working nodes. Both working and

failed nodes are detected through the finish table. The application's execution time depends on the number of the failed nodes, the amount of work which has been completed and the number of the remaining nodes which will do the recovery. In case transient faults have occurred, then, if the recovery of the failed nodes has already started and the failed nodes finally communicate with the master, the results must be taken into consideration, forcing the recovery process to stop, summing the results and terminating the application. In order to achieve this goal in a minimum time, three different checkpoint threads for the master have been created *(MThread, MThread-Single, MThread-Multiple)*. Each one of these threads is activated according to the conditions taking place and only one can be active at any time. Moreover, a continuous checking takes place for possible messages that may be received from nodes that were considered as failed, till the end of the recovery.

The fault-tolerant parallel algorithm goes as follows:

1. The search file is divided according to the total nodes of the cluster.
2. Worker nodes seek to the designated part of the search file according to their ID taken from MPI, while master always takes on the last part of the search file.
3. Workers start the checkpoint thread in order to send their critical values to master, while master starts the checkpoint thread (MThread) in order to receive workers' values and create the checkpoint and the finish tables.
4. If (rank==MASTER) search for pattern occurrences into the last part of the search file.

4.1. After search phase completes, wait until checkpoint thread completes as well.

4.2. Check the finish table for possible node failures.

4.2.1. If no fails have been detected, then sum the occurrences that each node found using information from the checkpoint table and send finish messages to workers.

4.2.2. If failures are detected then,

4.2.2.1. If only one failure has been detected, utilize checkpoint table's information for the progress of the failed node; set file pointer to the last known position; start MThread-Single in order to check whether or not the failed node finally returns; and if not, then complete the recovery and finalize the application.

4.2.2.2. If more than one failures have been detected, then select from the finish table the available nodes; send them recovering messages concerning the last known progress of the failed nodes, by utilizing information from the checkpoint table; start MThread-Multiple in order to check whether or not the failed nodes finally return; renew the checkpoint table; start recovering the first failed node and go to step 4.1.

5. If (rank!=MASTER) search for pattern occurrences into the corresponding part of the search file.

5.1. After search phase comes to an end, wait for a message from the master node.

5.1.1. If a finish message arrives, then exit.

5.1.2. If a recovering message arrives, then change your ID with the one of the node to be recovered, set the file pointer to the last known position of the failed node, start the checkpoint thread for sending key values to master, finish the remaining work and go to step 5.1.

3.1. Implementation of the MThread

After setting the file pointer to the start of its corresponding part, in order to begin the searching procedure, master creates the checkpoint table by starting the checkpoint thread. This thread is responsible for receiving workers' messages containing the key values of the file pointer, which indicates the searching progress, as well as the pattern occurrences already found; it runs concurrently with the main thread which implements the searching phase. The message receiving is accomplished with the non blocking routine MPI_Irecv of MPI, so as to overlap communication with computation. Thread receives workers' messages, one at a time and ignores repeated data that may be sent from a worker node, whose searching phase may advance slower than the checkpoint threshold. The thread ends in two cases. The first one is the case where all workers successfully finish their work indicating that no faults have occurred and this is done by checking the finish table every time a message arrives. The second one is when some (or even all) workers stop sending messages to the master node prior to finishing their work, thus indicating the presence of failures either permanent or transient due to network delay. When no more messages are received, master thread waits for a certain period of time *Twait* for workers' return and after that the thread ends:

do{
wait for message from workers
i. if a message arrives and the values for the particular worker are updated compared to those in the checkpoint table then renew values. Mark the finish table for workers which completed their work
ii. else, wait for time equal to Twait and if still no messages arrive then end the thread
} while (all workers have not yet finished)

3.2. Single failure detection - MThread-Single

In case a single failure is detected, its recovery will be made by master, as it was previously described. In such a case master starts the MThread-Single in order to wait for the failed node's possible return before the termination of the recovery procedure. It is a variation of the MThread with the main difference being that this thread does not end after time *Twait,* in case no messages are received, but it is active until the completion of the recovery. Thus, maximum time is given to the failed worker for possible return. As previously mentioned, this is a transient failure situation during which a worker node stops communicating with the master for a period of time. Even though the searching phase may proceed normally for the worker node, messages are not sent to the master node and thus the worker node is considered as failed. So, if at any time before the termination of the recovery the node sends its critical values to the master, then the master should be able to receive these values, terminate the recovery, set the occurrences found during the recovery procedure, to zero, in order to avoid wrong results, and then finalize the thread.

3.3. Multiple failures detection - MThread-Multiple

In case multiple failures are detected, the first failed node will be recovered by the master and the others by the available nodes of the cluster. After the allocation of the work that each node has to perform, the master begins the MThread-Multiple in order to wait for

the failed nodes' possible return before the termination of the recovery procedure. This thread is also a variation of the MThread and has been created in order to wait for possible messages from the failed nodes. In case all recovering nodes except the master have finished their work, the finish table would be complete and the MThread would end, eliminating the case of the return of the node being recovered by the master. So, if nodes that are recovered finally return, before the one which the master tries to recover, the thread does not end, but it remains active until the end of the recovery that master performs. If the failed node finally returns, then its values are registered to the checkpoint table, the recovery stops, the occurrences found during the recovery procedure in the master are set to zero, in order to avoid wrong results, and then the thread is finalized.

3.4. Implementation of the Worker_Thread

Worker_Thread is responsible for sending key values at regular intervals to the master, concerning the searching phase until it completes. If master thread stops receiving these messages, either due to permanent node fault, or due to network delay, the particular node is considered as failed. Again, note that, in the case of network delay, although master may have stopped receiving messages, the workers' searching phase continues to proceed into their main thread.

3.5. Generation of faults

Although our application supports failures at any point in the execution, having constant failure points proves to be the most practical and reproducible approach for performance measurements. Thus, failures are performed manually by introducing a piece of code with a SLEEP statement into the Worker_Thread, which emulates the presence of faults, by stopping the message passing to master node. Through this piece of code, it is possible to select which nodes will be considered as failed and the exact time these fails will occur. Also, it is possible to select if nodes that are considered as failed will finally return, emulating any possible network delays that may exist and, thus, generating permanent or transient failure situations.

4. A LOAD BALANCING ALGORITHM FOR HETEROGENEOUS CLUSTER ENVIRONMENTS

Herein, a load balancing algorithm is presented for the previous application to achieve exceptional performance when executed in heterogeneous clusters. Taking into account that the same work is assigned to all nodes, the previous algorithm is effective only when executed in homogeneous clusters. In case of heterogeneous clusters, from the perspective of computational power, the load balancing algorithm that is presented assigns work to nodes in proportion to their processing capacity. Therefore, the same working time is achieved for all nodes, instead of the same working load that was applied with the previous workload distribution. The goal of the algorithm is the effective utilization of all cluster nodes according to their processing power. For the implementation of the algorithm the first step that one has to do, is to evaluate the processing power of each node regarding the total cluster's processing power; this is done by calculating weights for each node of the cluster.

The total algorithm can be analyzed as follows:

1. Calculation of weights for each node of the cluster.

In order to calculate the weights for each node, one way is to evaluate the total time that each node needs to execute the sequential version of the particular application. Another way is to count the checkpoints needed for each node to complete the part of work that corresponds to it, by executing the parallel version of the algorithm using all nodes of the cluster. Both ways have been experimentally evaluated and the weights that computed came up to be alike. By using the previous methods the weights w_i for each node are computed regarding the fastest node as follows:

$w_i = \frac{t_j}{t_i}$, where t_j is the minimum time evaluated and belongs to the fastest node, while t_i is the time that each node needs to execute the sequential application, or the time that each node needs to execute the part of its work in the parallel application derived from the number of checkpoints.

2. Normalization of weights in the unit in order to readjust the weights in a percentage scale.

The normalization of the previous weights in the unit is conducted using two steps. The first step is to sum up all the previous weights and the second step is to divide each weight with the previous calculated sum for each node:

α. $w_{sum} = \sum_1^n w_i$, where n is the total number of cluster's nodes

b. $w'_i = \frac{w_i}{w_{sum}}$, for each node of the cluster, where w'_i represents the weights normalized in the unit.

3. Calculation of the partial size of the search file's total size that each node will process according to its weight.

The partial size of the search file's total size, that each node will process, is calculated from the product of the corresponding to each node normalized weight with the total search file size:

for i=1 : (mysize-1)

$fsz_i = w'_i$ * search_file_size

where mysize is the total number of processes, fsz is the part of the search file size that each node will process and search_file_size is the total search file size in which the pattern matching will take place.

4. Calculation of the limits between which the searching phase will take place for each node whose partial size has been calculated in step 3, starting from the beginning for the first node and assigning the last part of the search file to master node according to its weight.

Each node searches for patterns starting from the beginning of the part that corresponds to it and proceeding accordingly to its particular partial size. Calculation of the limits in which the searching phase will take place for each node according to its partial size is done as follows:

for i=1:(mysize-1)

$limits_i = limits_{i-1} + fsz_i$

beginning from zero point, meaning the start of the file, for the first node and assigning the last part to master node.

5. EXPERIMENTAL RESULTS

The platform where all the experiments were performed is a 16-node cluster of commodity workstations. The nodes are interconnected with a Gigabit network Ethernet. Many of the nodes have different hardware characteristics, thus, forming a heterogeneous environment. The operating system installed on each node is MS Windows XP and the message passing implementation used is MPICH-NT. In order to utilize threads, the applications were implemented in the C++ programming language.

The hardware characteristics of each node of the cluster are presented in table 1:

NODE ID	CPU	RAM
node1	Intel Pentium 4 - 2.8 GHz	1GB
node2-4	AMD Athlon - 1.7 GHz	1GB
node5	Intel Pentium 4 - 2.4 GHz	1GB
node6	Intel Pentium 4 - 1.8 GHz	1GB
node7-8	Intel Pentium 4 - 2.8 GHz	1GB
node9	Intel Pentium 4 - 3 GHz	1GB
node10-16	Intel Pentium 4 - 2.8 GHz	1GB

Table 1: Hardware characteristics of the cluster

The normalized weights in the unit that were calculated for each node of the cluster are presented in the histogram of Figure 1. The test files that were used as the search files are from the DNA sequencing area. DNA sequencing is an active and very important field of research, with databases constantly evolving, containing an increasing amount of data. The search files were downloaded from the site of GenBank, a genetic sequence database that belongs to the National Institute of Health (cf. 6). These files were chosen because of the importance of pattern matching in DNA sequencing and the large size they have.

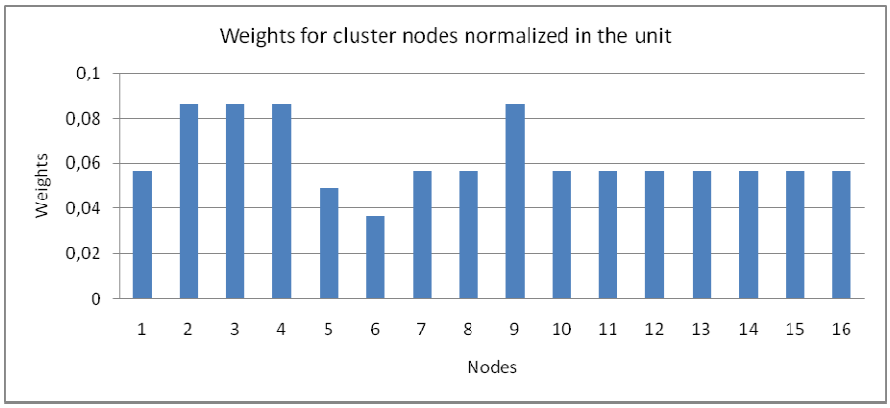

Figure 1: Normalized weights for cluster nodes

The application versions, whose performance were evaluated and will be compared in this section, will be referred from now on as:

BF: it is the parallel pattern matching version without fault-tolerance or load balancing support, targeting at homogeneous clusters.

BF-LB: it is the BF version with load balancing support targeting at heterogeneous clusters.
BF-FT: it is the BF version with fault-tolerance support for homogeneous clusters.
BF-FT-LB: it is the BF-FT version with load balancing support for heterogeneous clusters.

The initial time T_{wait} that master thread *(MThread)* waits, for versions *BF-FT and BF-FT-LB,* - if no worker messages arrive – has been chosen to be 10 seconds in order to compensate with any network delays. After this period of time, if still no messages arrive, the thread terminates. On the other side, workers are arranged to send their critical values to master every one second. Thus, there is enough information for recovering failed nodes, without causing network congestion.

The cases that we will examine are separated into two major categories. The first one is the execution of the parallel pattern matching application with the absence of faults and the second one is the parallel pattern matching application where faults are introduced. In the first category, experimental results will be presented from the execution of the versions BF, BF-LB, BF-FT, BF-FT-LB for two different search files and for different number of processors used for solving the problem, in order to evaluate the performance of each version and the overhead that is introduced. In the second category, experimental results will be presented from the execution of BF-FT and BF-FT-LB versions with an increasing number of faults introduced. Several different scenarios will be presented concerning various numbers of permanent or transient faults, as well as the exact time faults occur. The results concern the large search file and the total number of cluster's nodes.

Herein, speedup diagrams for all versions without faults are presented, in order to evaluate each version's performance. The experimental results presented, in figure 2 and figure 3, include searching for occurrences of a 5Bytes pattern into two search files, one of size 286MB and another one of size 3,16GB. Patterns of size up to 20 Bytes have been tested without noting any substantial differences in the execution time. Thus, at least for small patterns, their size is immaterial to the performance achieved.

Task mapping to processors can be configured through machinefile of MPI implementation and herein it is done from *node16* -which has the role of the master node- to *node1*. Thus, all nodes except *node16* are worker nodes. The execution time is the average time of several executions. For the versions without load balancing support, the speedup is almost linear only when up to 8 nodes are utilized. For more than 8 nodes the environment becomes heterogeneous, for the particular task mapping; thus, the utilization of the added nodes does not increase the speedup, due to the fact that the total execution time depends upon the processing capacity of the slowest nodes.

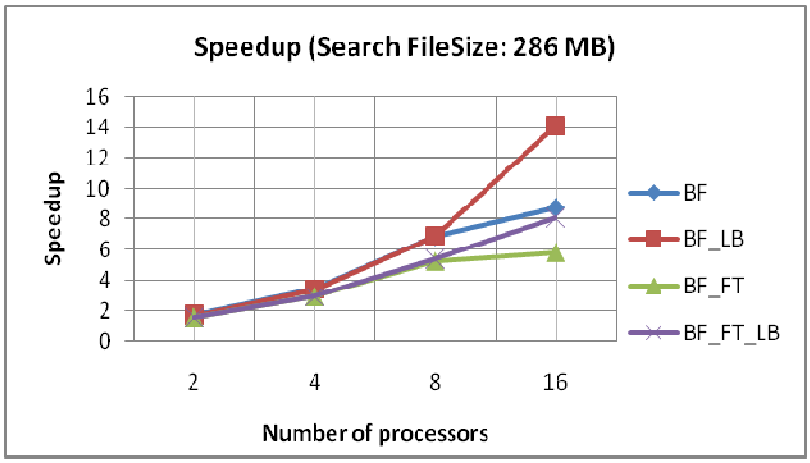

Figure 2: Speedup diagrams for all versions in the case of a small search file

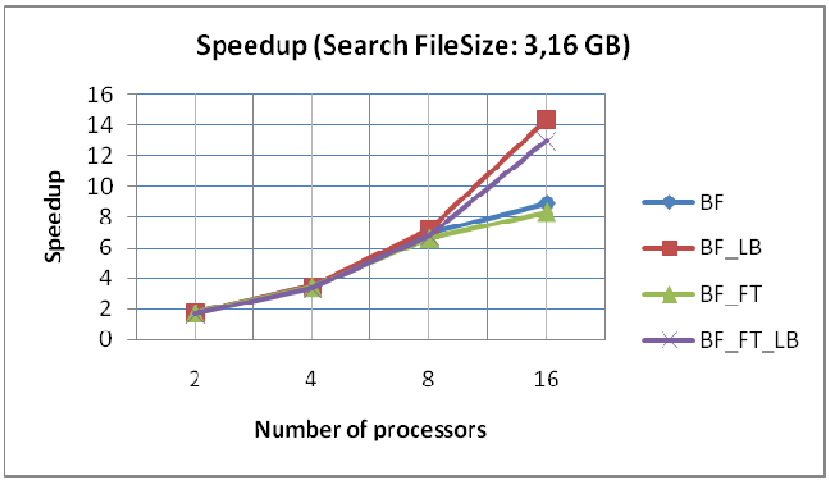

Figure 3: Speedup diagrams for all versions in the case of a large search file

According to the experimental results shown in figure 2 and regarding the fault-tolerant algorithm, it can be observed that the overhead is bigger in the case of the small search file, due to the fact that communication overlaps computation. In case of the large search file, in figure 3, it can be observed that the overhead decreases, indicating only benefits from the added fault-tolerance support. The average overhead for the BF-FT and the BF-FT-LB versions without any faults introduced, was evaluated to 3.6745 seconds compared to the versions without fault-tolerance support and for the case of the large search file, with all nodes of the cluster being utilized.

From the experimental results and regarding the load balancing algorithm, it can be observed that, when all nodes are utilized, there is a significant decrease in the execution time. Thus, speedup is increased from 8,890 to 14,336 for the BF-LB version as compared to the BF version, achieving a 37,99% gain in the total execution time, and from 8,336 to 13,076 for the BF-FT-LB version as compared to the BF-FT version, achieving a 36,25% gain in the total execution time.

The execution time for all versions and for the case of the large search file, utilizing all nodes of the cluster is presented in the histogram of figure 4.

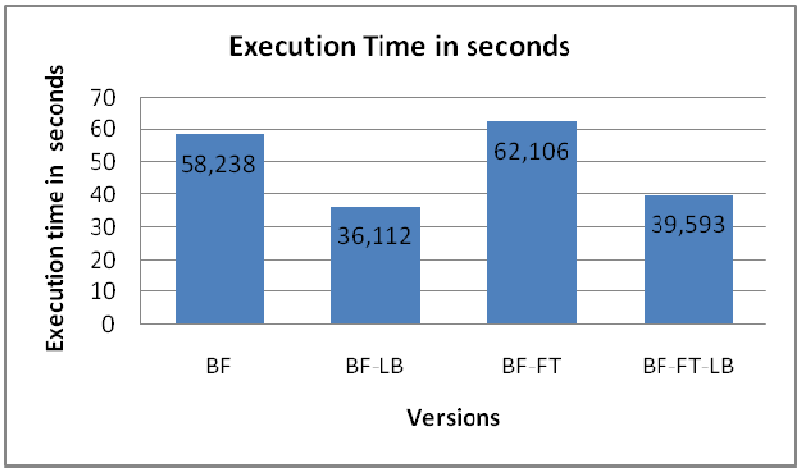

Figure 4: Execution time for all versions in the case of the large search file utilizing all nodes of the cluster

The execution time of the serial brute force pattern matching algorithm has been evaluated to 517,708 seconds.

5.1. Experimental results for the case of permanent node faults

Herein, experimental results will be presented for the BF-FT and BF-FT-LB versions, using the large search file, when permanent faults occur. Two characteristic cases will be examined. The first is the case where faults are introduced at the beginning of the application's execution, while the second is the case where faults are introduced 20 seconds after the beginning of the application's execution, when a part of the work has already been performed by all nodes. In the histogram of figure 5 is presented the execution time that each application needed to complete for the two cases and for an increasing number of faults introduced. It follows table 2, in which all these results are grouped. Faults of 1, 2, 8, 9, 15 nodes are introduced, where in the case of 15 faults the recovery of all failed nodes is performed by the master node, the only one that is considered as a working node. In case master node fails, then the whole application will also fail. In order to deal with this, additional hardware and software approaches must be adopted in order to replicate the master node, offering high availability functionality (cf. 5).

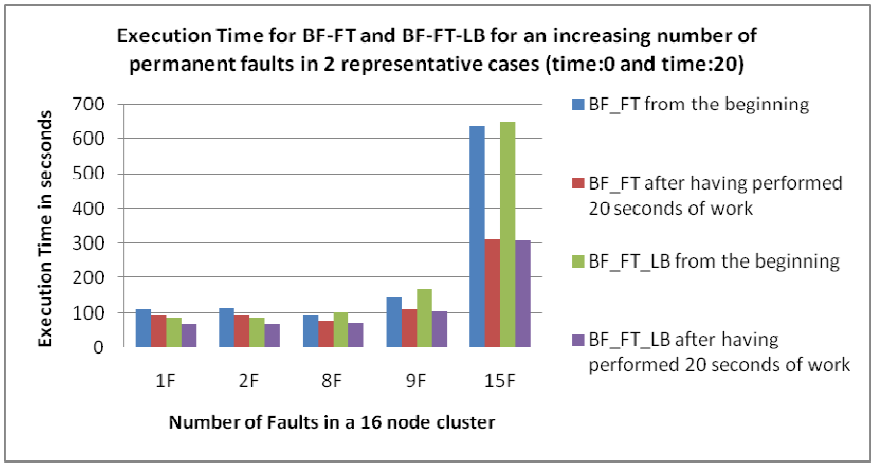

Figure 5: Execution time for BF_FT and BF_FT_LB versions for an increasing number of permanent faults

Without Load Balancing *(BF-FT)*	**Time in seconds**	**With Load Balancing *(BF-FT-LB)***	**Time in seconds**
No FAULTS	*62,106*	**No FAULTS**	*39,593*
# of FAULTS		**# of FAULTS**	
From the beginning (time:0)		**From the beginning (time:0)**	
1F(1)	112,091	1F(1)	86,304
2F(1,8)	113,035	2F(1,8)	86,721
		2F(1,2)	104,627
8F(1,2,3,4,5,6,7,8)	94,516	8F(1,2,3,4,5,6,7,8)	102,942
9F(1,2,3,4,5,6,7,8,9)	145,796	9F(1,2,3,4,5,6,7,8,9)	168,026
15F(All except 16)	637,716	15F(All except 16)	649,328
After having performed 20 seconds of work		**After having performed 20 seconds of work**	
1F(1)	91,925	1F(1)	65,889
2F(1,8)	92,972	2F(1,8)	66,086
		2F(1,2)	72,736
8F(1,2,3,4,5,6,7,8)	78,809	8F(1,2,3,4,5,6,7,8)	72,720
9F(1,2,3,4,5,6,7,8,9)	110,844	9F(1,2,3,4,5,6,7,8,9)	104,346
15F(All except 16)	313,412	15F(All except 16)	309,449

Table 2: Analytical and grouped results of figure 5

Due to the heterogeneity of the nodes, the execution time for each scenario depends on which nodes fail and which nodes are available in order to make the recovery of the failed ones. For our scenarios, the nodes which fail are marked into brackets in the above table. The nodes that recover the failed ones are selectively chosen, beginning from the master node, for the recovery of the first failed node, and finding the available ones in order to assign them the task of recovering the rest of the failed nodes.

The main objective of the applications is to return correct pattern occurrences, for each of the above mentioned scenarios and this must be done in acceptable time limits. The experimental results confirm the above goal, showing that the application can effectively handle up to (n-1) faults.

The total execution time depends upon the time that each application requires to complete without any faults occurrence, plus the time defined for the master node to wait for any message being received, when communication between workers and master node stops; plus the time that the slowest recovery node needs for completing the remaining work of the failed node recovering:

$$\mathbf{T} = \mathbf{T_{nf}} + \mathbf{T_{wait}} + \mathbf{max}\{\frac{\boldsymbol{w_{r1}}}{\boldsymbol{w_{f1}}} * \boldsymbol{Tf_{1remain}}, \frac{\boldsymbol{w_{r2}}}{\boldsymbol{w_{f2}}} * \boldsymbol{Tf_{2remain}}, \cdots, \frac{\boldsymbol{w_{ri}}}{\boldsymbol{w_{fi}}} * \boldsymbol{Tf_{iremain}}\}$$

where T is the total execution time, T_{nf} is the execution time when no faults occur, T_{wait} is the waiting time, which has been selected to be 10 seconds for all the experiments, w_{ri} is the weight of the recovery node, w_{fi} is the weight of the failed node that is recovered by the node whose weight is w_{ri} and $Tf_{iremain}$ is the remaining time that the failed node would need to complete its work if it had not failed. The index i refers to the ID of the failed nodes and to the ID of the nodes that make the recovery, accordingly.

When up to 50% of nodes present a failure, then their recovery takes place concurrently from the other 50% of nodes; consequently the worst total execution time increases similarly to the case where only one node fails. In case more than 50% of nodes fail, then due to the fact that there are not enough remaining nodes to recover the failed ones concurrently, the total execution time increases, depending on how many of the nodes are available to be utilized for the recovery.

5.2. Experimental results for the case of transient node faults

Herein, experimental results will be presented for the BF-FT and BF-FT-LB versions, using the large search file, when one and two transient faults occur. Transient faults, are faults detected due to network delay. Although the recovery process starts, worker nodes finally communicate with the master node sending their key values. Similar to the previous section, two typical cases will be examined. The first is the case where faults are introduced at the beginning of the application's execution, while the second is the case where faults are introduced 20 seconds after the beginning of the application's execution, when a portion of the work has already been performed by all nodes.

The nodes that are detected as failed, but finally return, are marked in brackets in table 3. Moreover, the exact time the nodes return is also referred. As it can be observed, while the recovery of the considered as failed nodes (due to large network delay) may has started, if nodes finally return then the application is terminated with a total execution time depending upon the return time of the last node. The total number of pattern occurrences is correctly computed in every case.

Without Load Balancing (BF-FT)	**Time in seconds**	**With Load Balancing (BF-FT-LB)**	**Time in seconds**
# of FAULTS		**# of FAULTS**	
From the beginning (time:0)		**From the beginning (time:0)**	
1F(1) : returns at the 75th second	76,779	1F(1) : returns at the 55th second	57,141
2F(1,8) : return at the 75th second	76,215	2F(1,8) : return at the 55th second	56,769
After having performed 20 seconds of work		**After having performed 20 seconds of work**	
1F(1) : returns at the 75th second	76,719	1F(1) : returns at the 55th second	57,376
2F(1,8) : return at the 75th second	76,939	2F(1,8) : return at the 55th second	56,814

Table 3: Experimental results for the case of transient node faults

5.3. Experimental results for the case that a recovering node also fails

In the following table the experimental results for a case a node that recovers a failed node also fails, are presented. More specifically, the case where two nodes are detected as failed and the recovery process starts is considered. The first failed node is recovered by the master node, while the second failed node is recovered by a worker node. While the worker node performs the recovery, it also fails and thus the remaining work is finally performed by the master node. The execution time that is marked concerns both BF-FT

and BF-FT-LB versions, when the initial faults occur a) at the beginning of the execution, and b) after having performed 10 seconds of work. The second fault, the one for the recovering node, also occurs after having performed 10 seconds of the remaining work.

Thus, from the experimental results of table 4, it can be observed that even when a recovering node also fails, the application is capable of completing the remaining work in acceptable time limits.

6. CONCLUSIONS

Herein, a fault-tolerant parallel pattern matching algorithm with load balancing support for homogeneous/heterogeneous clusters has been presented. The main intention was the return of reliable results in any case of faults, either permanent or transient, in acceptable time limits. In case of occurring faults, transparently to the user, the application can recover from any number of faults, provided that the master node is still up. As seen from the experimental results, the overhead of the fault-tolerant mechanism is relatively small when compared to the overhead the re-execution of the application would cause. Consequently, such an approach is highly recommended for long hour applications executed on commodity clusters, where faults are most likely to occur. Moreover, the load balancing algorithm for heterogeneous cluster environments presented herein, exhibited a significant increase in applications' speedup.

Without Load Balancing (BF-FT)	**Time in seconds**	**With Load Balancing (BF-FT-LB)**	**Time in seconds**
2 (+1) Faults(1,8,*15*) : *Initial faults* occur at the beginning of the application's execution *Second fault* occurs after having performed 10 seconds of work	141,913	**2 (+1)** Faults(1,8,*15*) : *Initial faults* occur at the beginning of the application's execution *Second fault* occurs after having performed 10 seconds of work	113,700
2 (+1) Faults(1,8,*15*) : *Inital fault*s occur after 10 seconds of the application's execution *Second fault* occurs after having performed 10 seconds of the remaining work	121,501	**2 (+1)** Faults(1,8,*15*) : *Initial fault*s occur after 10 seconds of the application's execution *Second fault* occurs after having performed 10 seconds of the remaining work	92,837

Table 4: Experimental results for the case a recovering node also fails

REFERENCES

1. Treaster, M. (2005). A Survey of Fault-Tolerance and Fault-Recovery Techniques in Parallel Systems, *ACM Computing Research Repository*, v.501002, pp. 1-11.
2. Gropp W. & Lusk E. (2002). Fault Tolerance in MPI Programs, *Special issue of the Journal High Performance Computing Applications (IJHPCA)*, v.18, pp. 363-372.
3. Efremides O. & Ivanov G. (2006). A Fault-Tolerant Parallel Text Searching Technique on a Cluster of Workstations, *Proceedings of the 5th WSEAS International Conference on Applied Computer Science*, pp. 368-373.

4. Elnozahy E. N., Alvisi L., Wang Y.-M. & Johnson D. B. (2002). A survey of rollback-recovery protocols in message-passing systems, *ACM Computing Surveys*, v.34, pp. 375–408.
5. IEEE Task Force on Cluster Computing. http://ieeetfcc.org/high-availability.html
6. Genbank: National Institutes of Health genetic sequence database. http://www.ncbi.nih.gov/Genbank
7. Charras C. & Thierry L. (2004). Handbook of Exact String Matching Algorithms: King's College Publications.
8. Camargo R. Y., Goldchleger A., Kon F., & Goldman A. (2004). Checkpointing-based rollback recovery for parallel applications on the InteGrade grid middleware, *Proceedings of the 2nd Workshop on Middleware for Grid Computing (ACM)*, v.76, pp. 35-40.