27

Implementation of Fault Tolerance Techniques for Grid Systems

Meenakshi B. Bheevgade and Rajendra M. Patrikar
Visvesvaraya National Institute of Technology Deemed University, NAGPUR.
Visvesvaraya National Institute of Technology,
NAGPUR, MAHARASHTRA STATE,
INDIA

1. Introduction

In the modern era of super-computing, grid of computing nodes has emerged as a representative means of connecting distributed computers or resources scattered all over the world for the purpose of computing and distributed storage. Most of the organizations use latest technologies to form a grid. Complex scientific problems in science and engineering run for a long time and it becomes important to make them resistant to failures in the underlying hardware and infrastructure. Grid Computing System has used for execution of the application, which needs more time. During parallel application, the computation cannot complete if any node(s) failures(~~s~~) encountered. Therefore, fault tolerance has become necessity. The fault tolerant techniques usually compromise between efficiency and reliability of the node in order to complete the computation even in presence of failures. The goal usually is to preserve efficiency hoping that failures will be less. However, the computational resources have increased in Grid but its dynamic behavior makes the environment unpredictable and prone to failure. A major hurdle parallel applications facing today is appropriate handling of failures that occur in the grid environment. Most application developers are unaware of the different types of failures that may occur in the grid environment. Understanding and handling failures imposes an undue burden on the application developer already burdened with the development of their complex distributed application. Thus, automatic handling of faults by grid middleware is necessary. This strategy makes this functionality transparent to the application users. This would enable different data-intensive grid applications to become fault-tolerant without each having to pay a separate cost. In this chapter, we discuss the various strategies of fault tolerance. The watchdog timer algorithm, a popular method in embedded systems, has been used to bring in fault tolerance in cluster and grid environment. An implementation detail of the watchdog timer like strategy in cluster environment is discussed in detail. This method requires a modification of application program. This methodology was further developed in which program state was collected at regular interval so as to start the program at appropriate time. The technique allows the execution of long-running parallel application even in the presence of node failure. The Watchdog Timer method is used in our work to

take care of hardware as well as software faults. The implementation of this method helped us to detect the failure of the node. In the embedded systems, timer resets the system if fault occurs. However, for the parallel computation performance applications this strategy may prove to be costly and may not be possible every time. It is desirable that the application running on the faulty node, which is usually a part of the program spawned by the main routine, should continue to run on healthy node after failure detection. The strategy is that the application resumed on the newly added node from the last saved data. In this method the parallel application need to modify for collecting the state of the intermediate steps. This strategy could be incorporated in the application itself. However, this may increase the complexity of the application and may slow down the application. Thus the long running applications need not start again from the beginning which is usually very costly. This task is done by the master node and we assume that this node has a very high reliability. (In grid, there is no domain controller. All nodes are treated as of same hierarchical level. The node on which I execute and distribute the task from it, I refer that node as Master node for the simplicity.) Since cluster and grid layers were different for the middleware implementation, different sets of application were developed. The collection of state (i.e. data) of parallel application at the middleware of cluster and grid layer using different sets of application was referred as "Sanchita" (meaning Collection in Sanskrit language also abbreviated from Search All Nodes for Collection of Hidden data (Process State) by Implementing The Algorithm) Fault Tolerant Technique. The technique helps to take care of different types of failures results to complete the task of computation and to achieve fault tolerance in cluster and grid environment. These applications were integrated to make it transparent to user.

2. Review

Lot of work has been done on fault tolerant mechanisms in distributed parallel systems. The focus of that work is on the provision of a single failure recovery mechanism. The different types of failures have been detected called as failures detection and failures recovery mechanisms have been applied. In perspective of parallel programs, there are vendor implementations of check-point/restart for MPI applications running on some commercial parallel computers. A checkpoint/restart implementation for MPI at NCCU Taiwan uses a combination of coordinated and uncoordinated strategies for checkpointing MPI applications. It is build on top of the NCCU MPI implementation uses Libtckpt as the back-end checkpointer. A local daemon process coordinates checkpointing of the processes running on the same node, while processes on different nodes are checkpointed in an uncoordinated manner using message logging (Sankaran, et al., 2005). A limitation of the existing systems for checkpointing MPI applications on commodity clusters is that they use MPI libraries, which primarily serves as research platform in most of the cases. Secondly, the checkpoint/restart system was tightly coupled to a specific single-process check-pointer. Since single-process check-pointers usually support a limited number of platforms, limits the range of systems on which this technique could be applied. A transparent checkpoint-restart mechanism for commodity operating systems had evaluated that checkpoints and restarts multiple processes in a consistent manner. This system combines a kernel-level checkpoint mechanism with a hybrid user level and kernel-level restart mechanism to leverage existing operating system interfaces and functionality as much as possible for transparent checkpoint-restart (Laadan & Nieh, 2007). The fault tolerance problem in term of

resource failure had addressed in (Nazir, Khan, 2006). The author devised a strategy for fault tolerant job scheduling in computational grid. This strategy maintains history of the fault occurrence of resource in Grid Information Service (GIS). Whenever a resource broker has job to schedule, it uses the Resource Fault Occurrence History (RFOH) information from GIS and depending on this information use different intensity of check pointing and replication while scheduling the job on resources that have different tendency towards fault acceptable service. The fault tolerance function is to preserve the delivery of expected services despite the presence of fault-caused errors within the system itself and described by the author Avizienis. (Avizienis, 1985). Errors had been detected, corrected, and permanent faults were located and removed while the system continues to deliver acceptable service. Execution of SPMD applications in a fault tolerant manner had achieved using check pointing or replication method by Weissman (Weissman, 1999). For the purposes of a direct quantitative comparison, a simple checkpoint model had assumed in which each SPMD task saves its portion of the data domain on disk at a set of pre-determined iterations. Abawajy et al. (Abawajy, 2004) define resource as any capability that must be scheduled, assigned, or controlled by the underlying implementation to assure non-conflicting usage by processes. Scheduling policies for Grid systems can be classified into space sharing (Thanalapati and Dandamudi, 2001) and time-sharing. It is also possible to combine these two types of policies into a hybrid policy to design an on-line scheduling policy. Tuong designed a framework (Nguyen-Tuong, 2000), which enables the easy integration of fault-tolerance techniques into object-based grid applications. Using programming tools augmented with fault-tolerance capabilities, they have shown how applications had written to tolerate crash failures. A fault tolerance service was designed to incorporated, in a modular fashion, into distributed computing systems, tools, or applications. This service uses well-known techniques based on un-reliable fault detectors to detect and report component failure, while allowing the user to tradeoff timeliness of reporting against false positive rates (Stelling, at.al, 1999). The authors in (Liang, et.al, 2003) described the approach from user viewpoint of Grid and considered the nature of Grid faults across the board based on thread state capturing mechanism, an exception handling method and mobile agent technology. Globus has become the de facto standard for grid computing. The Globus toolkit consists of a set of tools and libraries to support grid applications. Fault tolerance approaches in grid systems had commonly achieved with checkpoint-recovery and job replication described in [(J.B.Weissman and Womack, 1996),(Abawajy, 2004), and (Townend, 2004)] which create replicas of running jobs and hoping that at least one of them succeeds in completing the job. The authors in (J.B.Weissman and Womack, 1996) introduced a scheduling technique for a distributed system that suffers from increased job delays due to insufficient number of remote sites to run the replicas.

Fault tolerance in grid environment was achieved by scheduling jobs in spite of insufficient replicas (Abawajy, 2004). This approach requires at least one site to volunteer for running the replica before the execution can start.

The author Townend had submitted jobs replicas to different sites that return the checksum of the result. (Townend, 2004), the checksums received from various sites then compared to ensure whether majority results were the same, in order to avoid a result from a malicious resource, which delays the retrieval of result until a majority had reached. Therefore, job delay increase may result not only from failures but also from the verification overhead. Wrzesinska proposed a solution that avoids the unneeded replication and restarting of jobs

by maintaining a global result table and allowing orphaned jobs to report to their grandparent incase their parent dies (Wrzesinska., et.al.). However, their approach is strictly for divide-and-conquer type of applications and cannot extend to environments where the sub-processes require communication. "Grid Workflow" leaves recovery decisions to the submitter of the job via user defined exception handling (Soonwook and Kesselman, 2003). Grid Workflow employs various task-level error-handling mechanisms, such as retrying on the same site, running from the last checkpoint, and replicating to other sites, as well as masking workflow level failure. Nonetheless, most task-level fault tolerant techniques mentioned by the authors Abawajy, Dandamudi, Weissman , and Womack. (Abawajy and Dandamudi, 2003, Weissman and Womack, 1996, and Abawajy, 2004), attempted to restart the job on alternative resources in the Grid in an event of a host crash. Hence, there is a need for complementing these approaches by improving failure handling at the site level, especially in a cluster-computing environment. In LinuxHA Clustering project, LinuxHA is a tool for building high availability Linux clusters using data replication as the primary technology. However, LinuxHA only provides a heartbeat and failover mechanism for a flat-structure cluster which does not easily support the Beowulf architecture commonly used by most job sites. OSCAR is a software stack for deploying and managing Beowulf clusters (Mugler, et.al, 2003) and (Thomas Naughton, *et al.,)*. This toolkit includes a GUI that simplifies cluster installation and management. Unfortunately, a detrimental factor of the Beowulf architecture is the single point of failure (SPoF). A cluster can go down completely with the failure of the single head node. Hence, there is a need to improve the high-availability (HA) aspect of the cluster design. The recently released HA-OSCAR software stack is an effort that makes inroads here. HA-OSCAR deals with availability and fault issues at the master node with multi-head failover architecture and service level fault tolerance mechanisms. PBS (Bayucan, et.al, 1999) and Condor (Tannenbaum, et.al, 2002) are resource management software widely used in the cluster community. While a HA solution [CCM, LinuxHA] for Condor job-manager exists, there a dearth of such solutions for the PBS job manager. The failure of Condor Central Manager (CM) leads to an inability to match new jobs and respond to queries regarding job status and usage statistics. Condor attempts to eliminate the single point of failure (i.e. the Condor CM) by having multiple CMs and a high availability daemon (HAD) which monitors them and ensures one of them is active at all times. Similarly, HA-OSCAR's self-healing core monitors the PBS server among other critical grid services (e.g. xinetd, gatekeeper etc), to guarantee the high availability in an event of any failure. The fault tolerance design evaluation (Object Management Group, 2001), and (Friedman and E.Hadad) has performed by means of simulation, experiments or combination of all these techniques. The reliability prediction of the system has compared to that of the system without fault tolerance. Physical parameters, quality of fault detection and recovery (Foster and A. Iamnitchi,2000), (Foster, et. al.,1999),(Sriram Rao,1999) and (Felber and Narasimhan,2004) algorithms has used as parameters in generating reliability predictions.

When degradation of performance takes place during recovery, reliability predictions need to be generated for various levels of performance. A different evaluation is needed when the reliability includes a specification of a minimum number of faults that are to be tolerated; regardless where in the system occurs. The fault tolerance techniques described in (Foster and lamnitchi, 2000, Foster, et.al. 1999) and (Lorenzo,et.al., 1999) many times results in lower performance because of the overheads required for reallocation of job which was subjected to gracefully degradation mode. A gracefully degradable system is one in which the user

does not see errors except, perhaps, as a reduced level of system functionality. Current practice in building reliable systems is not sufficient to efficiently build graceful degradation into any system. In a system with an automatic reconfiguration mechanism, graceful degradation becomes fairly easy to accomplish. After each error is detected, a new system reconfiguration is done to obtain maximal functionality using remaining system resources, resulting in a system that still functions, albeit with lower overall utility. A watchdog timer is a computer hardware-timing device that triggers a system reset, if the main program does not respond due to occurrence of some fault. The intention is to bring the system back from the hung state into normal operation. The most common use of watchdog timer is in embedded systems, where the specialized timer is often a built-in unit of a microcontroller. Watchdog timer may be more complex, attempting to save debug information onto a persistent medium; i.e. information useful for debugging the problem that caused the fault. The watchdog timer has been utilized for a fault tolerant technique in many serial applications. Usual implementation of this technique requires hardware implementation of timer/counter that usually interrupts CPU for corrective actions. Thus, although fault tolerant clusters are being researched for some time now, implementation of the fault tolerance architecture is a challenge. There are various types of failures which may occur in the cluster. Prediction of failure mechanism is very difficult task and strategies based on a particular failure mode may not help (Sriram,et.al.,1999, Toenend and Xu, 2003, & Felbar and Narasimham,2005). Typically, fault detection can be done by some sort of response and respond method (Toenend, 2003). Assuming that the node has been faulty and not responding, in such cases the software testing has been done to re-check that the node has been responding. If the fault is transient then rechecks are to be done and if repeated re-checks show the failure of the system then the system is re-booted. If the fault persists after rebooting of the system then the node implied to be faulty and has to be removed from the list of available nodes. For this, the two algorithms i.e. Fault detection algorithm and recovery algorithm have been studied and partially implemented in our work also. In the next section we define important terms which are used in implementation of all the fault tolerance mechanisms.

2.1 Process State, RPC and Compute Bound

Process state is the state field in the process descriptor. A process referred as a task, is an instance of a program in execution. The process is runnable, and it is either currently running or it is on a ready state waiting to run. This is the only possible state for a process executing in a portion of system memory in which user processes run; it can also apply to a process in kernel space (i.e., that portion of memory in which the kernel executes and provides its services) that is actively running. A signal is a very short message that can send to a process, or to a group of processes, that contains only a number identifying the signal. It permits interaction between user mode processes and allows the kernel to notify processes of system events. A signal can make a process aware that a specific event has occurred, and it can force a process to execute a signal handler function included in its code. Processes in user mode had forbidden to access those portions of memory that had allocated to the kernel or to other programs. RPC (Remote procedure call) - A system call is a request made via a software interrupt by an active process for a service performed by the kernel. The wait system call tells the operating system to suspend execution of that process until another process has completed. An interrupt is a signal to the kernel that an event has occurred, thus

result in changes in the sequence of instructions that has executed by the CPU. A software interrupt, also referred to as an exception, is an interrupt that originates in software, usually by a program in user mode. Process execution has stopped; the process is not running nor is it eligible to run. The state occurs if the process receives the SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU signal (whose default action is to stop the process to which they had sent) or if it receives any signal while it has being debugged.
Compute Bound - Computers that predominantly used peripherals were characterized as IO (input output bound) bound. A computer is frequently CPU bound implies that upgrading the CPU or optimizing code will improve the overall computer performance. CPU bound (or compute bound) is when the time for a computer to complete a task is determined principally by the speed of the central processor and main memory. Processor utilization is high, perhaps at 100% usage for many seconds or minutes. There are few interrupts such as those generated by peripherals. With the advent of multiple buses, parallel processing, multiprogramming, preemptive scheduling, advanced graphics cards, advanced sound cards and generally, more decentralized loads, it became less likely to identify one particular component as always being a bottleneck. It is likely that a computer's bottleneck shifts rapidly between components. Furthermore, in modern computers it is possible to have 100% CPU utilization with minimal impact to another component. Finally, tasks required of modern computers often emphasize quite different components, so that resolving bottleneck for one task may not affect the performance of another. For these reasons, upgrading a CPU does not always have a dramatic effect. The concept of being CPU bound is now one of many factors considered in modern computer performance. We have developed a code that helps us to know if failure is encountered. This is done by transferring data from all the nodes to master node (here we define master node, which spawns the jobs to other nodes) in Grid Environment.

3. Implementation

Today many idle machines are used to form the cluster so that parallel application can be executed on those clusters. We have used LAM/MPI software to make the cluster of the workstations and PCs. This LAM/MPI is open software and has been mostly used for research work. There is another popular standard for inter-cluster communication i.e. Globus Toolkit, which is used to form a grid. This tool also has been used mainly for research work. There are many commercial tools available to form the grid which are used for commercial purpose than research work we have not used them. In our work, we had build the two different clusters, later the two clusters were connected to form a grid (cluster-to-cluster communication), which handles run-time communications between parallel programs running on different clusters. In this work, first we had worked on cluster to find the failures encountered while executing parallel application. (Here the master node of each cluster had connected to form a grid.) The focus of the work was to design a method in cluster and grid environment such that it will help to detect failures if any, during computation of parallel application. In addition, a method is implemented which ignores the failures encountered and complete the parallel computation. The system is classified into two layers. These two layers are referred as –

1) Grid Layer (i.e. Upper Layer) and
2) Cluster Layer (i.e. Lower layer).

The two layers are as shown in figure 1a and 1b.

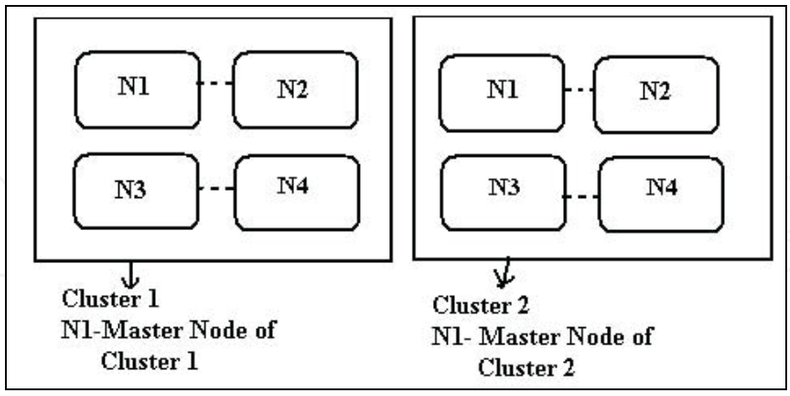

Fig. 1a Cluster Layer

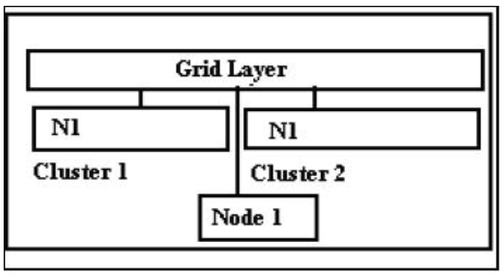

Fig. 1b Grid Layer

The grid layer of the system connects individual computers and clusters to form a grid as shown in figure 1b. The cluster layer connects the nodes in one group to offer their resources in the grid. These two layers were completely different but they were dependent on each other, if parallel application is executed in the grid environment. Due to these reasons different problems arises and different software solutions were used in both cases. In grid environment a Monte Carlo parallel application was executed from specific node say n, for our convenience, we referred that node as master node. There is no specific master/client concept in grid environment. From the user's point of view, both the layers are unique. The cluster-to-cluster communication approach allows programmers to collect the result of parallel program stored in one node to another node. In the cluster layer, we considered a set of nodes under control of one administrator. The nodes need to be detached from the grid during the routine work of the user (owner of the node since we were utilizing the idle time of the computer). Later on, re-connect the node when it is idle. The nodes cannot be a part of the grid when users work with it (nodes). The node may reboot incorrectly, it may killed the tasks it processed.

If only part[1] of the node had used during the user worked, the rest may still connected to the cluster to continue the computation.

[1]*(In a multi-processor node, the user may use one processor; the remaining processors may be allocated to the cluster for computation of parallel application. From remaining nodes available in a cluster, only few nodes can be part of the cluster and grid while users (owner(s) of the node(s)) for their personal work may use other nodes).*

In grid layer, as mentioned above, we connect two different clusters and few individual compute nodes with various security policies (authorization and authentication to access the node) as required in our project, we uses Globus Toolkit, for this purpose. This toolkit provides basic mechanisms such as communication, authentication, network information, and data access, resource location, resource scheduling, authentication, and data access. As the size of high performance clusters multiplies, probability of system failure also grows marginally, thereby, limiting further scalability. A system fault can cause by internal or external factors. For example, internal factors may include as specification and design errors, manufacturing defects, component defects, and component wear out. External factors may be included radiation, electromagnetic interference, operator error, and natural disasters and so on. The failure may also be due to hardware faults or software faults. As a result, it is almost certain for applications that run for a long time to face problem from either type of faults. Regardless, how well a system had designed, or how reliable the components are, a system cannot be completely free from failures. However, it is possible to manage failures to minimize its impact on a system.

We had implemented the fault tolerance technique (we called this technique as watchdog timer algorithm technique) for a cluster by writing routines on a master (server) node. The method implemented in our work includes re-checks to take care of transient faults included in the initial allocation phase. The cluster environment had booted by using only available nodes. In the beginning if the failure had been detected while allocating the task, then task had to reallocate again. After allocation, the state of entire parallel application had checked at specific intervals of time-period as it had done by hardware watchdog timer. In this phase, master node monitored an output data of the program at client node at specific time intervals. At the same time, the data had been stored on the reliable storage on the local disk as well as on the disk of the master node. The storage had done for all the nodes in the available list of the nodes. If any of the node had not reported or not responded when a master node send a message to all nodes in cluster environment, then that particular node had to be checked for **x** times. In this phase, the master node reallocates the task if failure had detected. This was done again to avoid the transient faults after certain intervals of time. Then the task had allocated once again after the application had terminated. Here, the checkpoint had done for x times because of any one reason tabulated in Table 1:

1	If at first call client's CPU utilization is 100%, it will unable to respond about its existence. In second call, the client may respond and update the status.
2	If link failure occurs due to network failure or due to some other reason, the Master node may try to reboot the client remotely. Therefore, in next call the client may respond and update the status.
3.	If it is not possible to reboot the client remotely, it has to be rebooted at the site, and if the rebooting is done in within stipulated time, it will respond in the next call.
4	Lastly, if any of the above options is not feasible and there is no response from client in the stipulated period, the master node will drop the current node from the list, add a new node and resume the job from the earlier check-pointed state on this new node.

Table 1. Reasons for the non-responding node

After the allocation phase starts, the client nodes had checked at regular intervals. If the allocation of job failed or the computation of the application failed due to network failure or node failure or may be some hardware component failure, then the flag had set to 1, indicates that the node will not be usable for the computation. Lastly, within stipulated time-period, if any of the above options was not feasible and there was no response from client node, the master node will drop the current node from the list, add another node and resume the job from the last best check-pointed state on another available or added node. It was possible to calculate the checksum that indicates correct behavior of a computation node. The watchdog timer algorithm will periodically test the computation with that checksum. The master node collects the data, which it determine the "state" of the client node when queried by the server. The algorithm given below in figure 2 is in the form of pseudo code.

- The Watchdog Timer Algorithm described below is implemented after LAM booting is done. Before Lam booting, a small procedure was implemented for creating the list of participating host.

```
for (i=1 to p) {
   do while (!execution_complete)
   {
     Monitor the application at regular intervals of time say t
seconds;
     collect the data into the client and send it to master when queried
by it;
     if (client !responding)
       { check for x times; // reason as given in table 1
        if (success)
          continue;
        else
         { drop the node;
           add a new node ;
           resume application on another available node say m in the queue.
         }
       }
   }
 }
```

Fig. 2. Watchdog Timer Fault Tolerant Technique

The actual implementation of the watchdog timer technique in our project work helped to detect the faults. The application had resumed on available node from the previous state saved data. Thus helps to recover the computation with the help of watchdog timer algorithm. Thus using the fault detection algorithm, recovery algorithm and watchdog theory we are able to improve the new algorithm. Thus, this method helped us to improve the reliability of the application although the performance degraded slightly because of computation overheads. Another drawback is that to store the data at specified interval of time one had to modify the parallel application(s). Thus to overcome the serious issue of

fault tolerance, another method i.e. Sanchita fault tolerant technique was implemented that collected the data at specified interval of time. The data had collected at compute bound rather than I/O bound. This data collection helped to restart the interrupted computation on another available node(s). This method is very effective approach for dealing with specified faults (like link failure, node failure). The state of the entire parallel application had saved into a file, and later it had transferred to the master node (the node on which the application is launched and distribution of task is done). In this work, the tool available in Linux called strace was used. A small program, was written to get the parent PID and child PID of the execution program. The tool helped to collect the information of system calls and to store the signals received by the process. This tool was used to store the relevant data of the parallel application at specified interval of time. The tool runs until the computation of parallel program does not complete. Then the relevant data was stored into a file in the local hard disk. The master node queried (send message) for the client nodes to check for the aliveness. The client node respond to the query and it send back the stored file to the master node. When a fault encountered, the application restarted from a recent last-saved point. In the beginning, to start with for applying the fault tolerant techniques, the experiment was done on one cluster only. Later the work had been carried out to find the problems encountered when the parallel application executed in grid environment. In this technique, the master node checks the aliveness of each node of the cluster at regular intervals of time by using a program. If the client node received the query done by master node, then client node replied to the master node about its existence. At the same time, it updated the master node with the status of the running process. If the client node did not respond before adding another node and resuming the process at that(another) node, the master node checks for x times and waits for response for approximately x+1 seconds. There can be number of situations in each call as explained above in table 1. The Sanchita fault tolerant technique was implemented and tested to store the last-saved state of the standard Monte Carlo application. The process state had resumed on the same node if any soft errors encountered. If hardware errors encountered such as, communication did not established or any physical problem of the node then the process resumed on another available node. The Sanchita technique applied at the middleware layer had given in the form of pseudo code as shown in figure 3. Each statement in the given pseudo code contained one or more than one program (or sub-routine).

```
Algorithm:(in the form of pseudo code)
While (execution in progress) do
  begin
     for (each node in cluster) do
        begin
            ping node for aliveness;
            if (alive) then
              begin
                collect checkpoint state from node;
                save in master node file;
              end;
            else
               begin
                  retry check for aliveness for k times;
                  if(success) then
                       begin
                            collect checkpoint state from node;
                            save in master node file;
                       end;
                  else
                       begin
                            drop node transfer computation to another node;
                       end;
               end;
        end;
  end;
```

Fig. 3. Algorithm used in Sanchita Fault Tolerant technique

4. Results

Watchdog timer algorithm was implemented in a SMP cluster based on the Linux. All nodes had the LAM/MPI software installed on it. The Monte Carlo application had tested in a parallel mode. The time required for the execution with and without applying the watchdog timer algorithm had shown in figure 4. The columns plotted in the graph pointed the following- The Column 1 pointed that the **w**atch**d**og **T**imer **a**lgorithm (WDTA) method had not applied and no faults were injected. The column 2 indicates data when WDTA method applied and no fault injected. This is to check that the WDTA code worked properly. The last column 3 pointed when WDTA method applied and one failure injected. The files are saved after interval of t seconds on individual nodes. The filename is referred as the node number followed by date followed by time. This time-stamped of the file helped to recognize from where the data has been collected. The files had collected on the master node when queried by it. If any failures encountered then the node had been checked for say x times about its existence. After checking if client node did not respond then the node had been treated as failed node. This failed node had replaced by another node. Some time to improve the performance of the grid; we had added few nodes in the cluster environment. These nodes were added only to check if performance had improved in the grid

environment. The performance had improved a lot. The programs were written using script language and 'C/C++' Language to add and drop the node. It also helped to improve the performance of the computation. The code was able to restart the application on another node. Further the work had been implemented in the grid environment. We had also taken care of the reliability of the grid environment. We had calculated the reliability of the grid environment to keep the nodes in spare. The work helped to resume the application from the last stored data that were based on the theories of the compute bound and process state. Thus, in the grid environment also, the status of the parallel application computed on the master node as well as on the client nodes (which are denoted by the rank(s) from '0' to n, where 'n' is the number of nodes) were collected. The same methodology is used here for the nomenclature of the filename i.e. the filename is referred as the node number followed by date followed by time. Thus date and time-stamped of the file helped to recognize from where the data had been collected. The work had been carried out to find if any failed node encountered. If encountered, then it was possible to resume the job on the new node and drop the earlier (failed) node. Further the second method i.e. Sanchita fault tolerance technique (SFTT) was implemented in the middleware level of the cluster and grid environment. When the application starts execution, the middleware level code takes care of the failures encountered. Using the Sanchita technique, the status of the computation was collected and shown in figure 5(few lines). Figure 6 shows the graph plotted with different combinations as shown in Table 2.

When SFTT code applied	Fault encountered	Node Added
No	No	Yes
No	No	No
Yes	No	Yes
Yes	No	No
Yes	Yes	Yes

Table 2. Data for SFTT

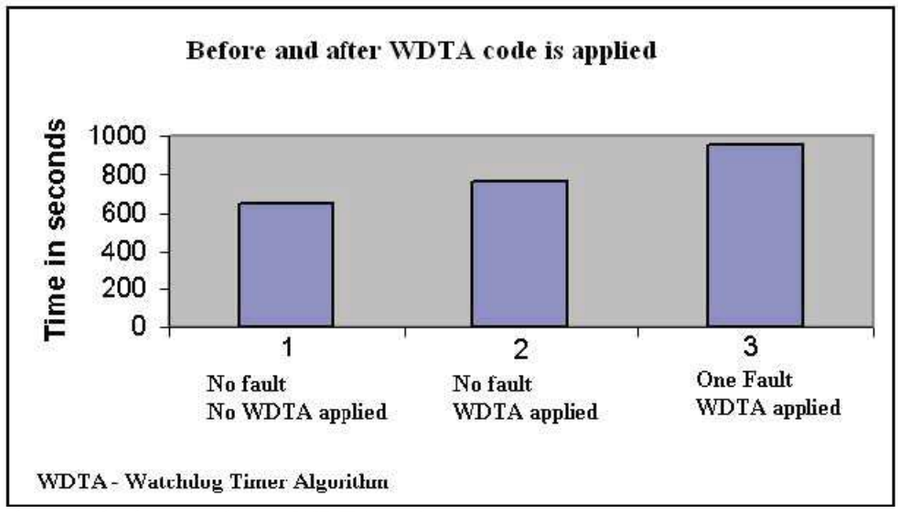

Fig. 4. Before and after applying WDTA code

```
read(3, 0xbffacf80, 80) = ? ERESTARTSYS (To be restarted)
--- SIGINT (Interrupt) @ 0 (0) ---
clone(child_stack=0,
flags=CLONE_CHILD_CLEARTID|CLONE_CHILD_SETTID|SIGCHLD,
child_tidptr=0xb7fe1708) = 5304
sigreturn() = ? (mask now [USR2])
read(3, 0xbffacf80, 80) = ? ERESTARTSYS (To be restarted)

    ---SIGCHLD (Child exited) @ 0 (0) ---
    read(3,"\0\0\0\0\0\0\0\20\0\0\0\6\244\4\10\1\0\0\0\360\337\354"...,
    80) = 8
    readv(3,{"n\353\377\277\0\0\0\0\0\0\0n\353\377\277\1\0\0\0\20\0"...,
    64}, {"\0\3\0\17\0\0\24\223\0\0\0\0\0\0\0", 16}], 2) = 80
    rt_sigprocmask(SIG_UNBLOCK, [USR2], NULL, 8) = 0
    rt_sigprocmask(SIG_BLOCK, [USR2], NULL, 8) = 0
    …….sa_family=AF_INET, sin_port=htons(32769),
    sin_addr=inet_addr("192.168.0.202")}, [16]) = 95sendto(6,
    "\0\2\310k\0\0\0\0", 8, n [6])recvfrom(6,
    "\0\0\0\1\0\2\310A\0\0\0\0@\0\0\7\0\0\0\0\0\0#\0\0\1\0"..., 8252,
    sin_addr=inet_addr("192.168.0.203")}, [16]) = 95sendto(6, "\0\2\310l\0\0\0\0",
    8, 0, {sa_family=AF_INET, sin_port=htons(32769),
    sin_addr=inet_addr("192.168.0.203")}, 16) = 8write(14, " 100 200 2470860"..., 35)
    = 35select(13, [3 5 6 7 10 11], NULL, [3 5 6 7 10 11], NULL) = 1 (in [6])recvfrom(6,
……..
```

Fig. 5. Sample File for tracing the parallel application running on the Grid System. (This is the few statements of the file)

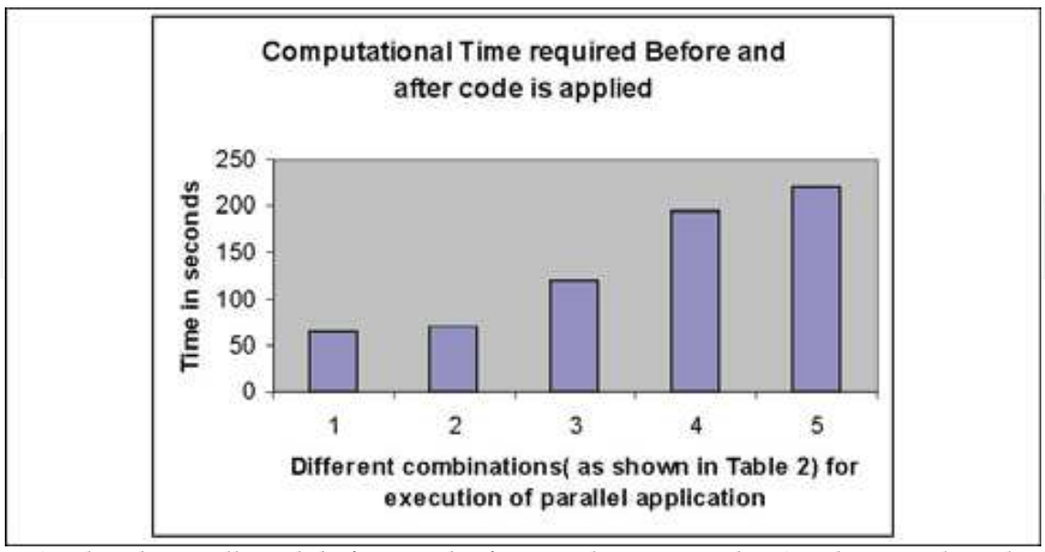

Fig. 6. The data collected before and after implementing the Sanchita Fault Tolerant Technique in cluster and grid environment.

In figure 6, Column 1 and 2 pointed the data when SFTT was not applied and no failures injected but one node was added. The time required after adding one node is less as compared to column 1. The third and fourth column pointed when Sanchita Fault Tolerant Technique (SFTT) was applied with no failure injected but node was added. These indicate that the SFTT code checks for every possibility of failures to complete the task. Column 5 pointed when SFTT was applied and one node failure encountered at the time of computation and another available node was added. From the last two columns, it was concluded that slightly more time was required to complete the computation.

5. Conclusion

The fault tolerant techniques are required in today's' cluster and grid environment since large applications are being run today for various applications and failure of single node can cause delays of days in execution and their by development also. In this chapter, two such techniques are described. These techniques are implemented in the application and in middleware layer of the cluster as well in the grid environment. Both the techniques i.e. watchdog timer and Sanchita were helpful in taking care of failures. However, in case of watchdog timer algorithm, the parallel application requires modification to get the status of the intermediate steps of the computation at specific intervals of time. It is the burden on the application developer. So in order to overcome this, Sanchita FT Technique was implemented. In this technique, states of the running jobs are collected at central (master) node and intermediate results are stored. The incomplete task (computation) due to node failure has resumed on another node using the latest computational data stored in a file. Thus, the application will take longer time but execution will be completed by resuming the task using the latest data stored on another node. Thus, system will gracefully degrade with failure. However, overall reliability will improve, which is needed for execution of many complex programs.

6. Acknowledgement

I would like to thank Professor Dr. C.S.Moghe for his kind support and suggestions during this work.

7. References

Oren Laadan & Jason Nieh, (2007). Transparent Checkpoint-Restart of Multiple Processes on Commodity Operating Systems, 2007 USENIX Annual Technical Conference

B. Nazir, T. Khan,(2006) *Fault Tolerant Job Scheduling in Computational Grid*, 2006 IEEE, pp 708-713

A. Avizienis, (1985) "The N-version Approach to Fault-Tolerant Software" -IEEE Transactions on Software Engineering - vol. 11 1985

J. H. Abawajy,(2004). Fault-Tolerant Scheduling Policy for Grid Computing Systems, 2004 IEEE.

J.H. Abawajy and S. P. Dandamudi.,(2003). Parallel job scheduling on multi-cluster computing systems, In *Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2003)*, Hong Kong, China, December 1-4 2003.

J. H. Abawajy,04-26-2004 "Fault-Tolerant Scheduling Policy for Grid Computing systems", 18th International Parallel and Distributed Processing Symposium, 04-26-04 Santa Fe, New Mexico

T. Thanalapati and S. Dandamudi.,(July 2001). An efficient adaptive scheduling scheme for distributed memory multicomputers, *IEEE Transactions on Parallel and Distributed Systems*, 12(7):758–768, July 2001.

A. Nguyen-Tuong, "Integrating Fault-Tolerance Techniques in Grid Applications"- Report, 2000

P. Stelling, et.al (1999). A Fault Detection Service for Wide Area Distributed Computations, 1999.

J. Liang, et. Al (2003). A Fault Tolerance Mechanism in Grid, 0-7803-5/04/2003, IEEE pp. 457-461.

J. B. Weissman and D.Womack.,(1996). "Fault tolerant scheduling in distributed networks"- Technical Report CS-96-10, Department of Computer Science, University of Virginia, Sep. 25 1996.

P. Townend, and J. Xu,(2003). "Fault Tolerance within Grid environment", Proceedings of AHM2003, http://www.nesc.ac.uk/events/ahm200 3/AHMCD/pdf/063.pdf, page 272, 2003

Soonwook Hwang and Kesselman, C , (2003). "Grid workflow: a flexible failure handling framework for the grid", High Performance Distributed Computing,2003. Proceeding 12th IEEE International Symposium, 22-24 June 2003,Pages 126 – 137.

John Mugler, *et.al.*(2003) "OSCAR Clusters", Proceedings of the Ottawa Linux Symposium (OLS'03), Ottawa, Canada, July 23-26, 2003. Thomas Naughton, *et al*. "The OSCAR Toolkit"

T. Tannenbaum, et. Al. (2002). Derek Wright, Karen Miller, and Miron Livny, "Condor -A Distributed Job Scheduler", *-Beowulf Cluster Computing with Linux*, The MIT Press, 2002. ISBN: 0-262-69274-0.

Ibeaus Bayucan, *et al. (1999)*, "Portable Batch System External Reference Specification", MRJ Technology Solutions, May 1999.

Ian Foster *et al*, "The Anatomy of the Grid: Enabling Scalable Virtual Organizations", International J. Supercomputer Applications, 15(3), 2001.

Ian Foster and A. Iamnitchi,(2000). "A problem – Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems", IEEE, p.4-13 2000.

Ian Foster, et. al (1999."A Fault Detection Service for Wide Area Distributed Computations", Cluster Computing, v.2 n.2,p.117-128, 1999.

Sriram Rao, et. al, (1999). Lorenzo Alvisi, Harrick M.V in , "Egida : An Extensible Toolkit For Low-overhead Fault-Tolerance, Fault-Tolerant Computing", Digest of Papers. Twenty-Ninth Annual International Symposium, p. 45-55, 1999.

J.G. Ganssle, (2004). "Great Watchdogs", V-1.2, Gaanssel Group, updated January, 2004. P.Townend and J. Xu,(2003) "Replication-based Fault-Tolerance in a Grid Environment", citeceer, 2003.

Pascal Felber, Proya Narasimhan,(2004) ,Member, IEEE, "Experiences, Strategies, and Challenges in Building Fault-Tolerant CORBA Systems", IEEE transactions on Computers , Vol.53, NO.5, May 2004.

Sriram Sankaran,et.al(2005) Parallel Checkpoint /Restart for MPI Applications. International Journal of High Performance Computing Applications, Vol. 19, No. 4, 479-493 (2005)

CCM, Adding High Availability to the Condor Central Manager, http://dsl.cs.rechnion.ac.il/projects/goxal/project_pages/ha/ha.html.

LinuxHA , Clustering Project, http://www.linuxha.net/index.pl.

Advanced Technologies

Edited by Kankesu Jayanthakumaran

ISBN 978-953-307-009-4

Hard cover, 698 pages

Publisher InTech

Published online 01, October, 2009

Published in print edition October, 2009

This book, edited by the Intech committee, combines several hotly debated topics in science, engineering, medicine, information technology, environment, economics and management, and provides a scholarly contribution to its further development. In view of the topical importance of, and the great emphasis placed by the emerging needs of the changing world, it was decided to have this special book publication comprise thirty six chapters which focus on multi-disciplinary and inter-disciplinary topics. The inter-disciplinary works were limited in their capacity so a more coherent and constructive alternative was needed. Our expectation is that this book will help fill this gap because it has crossed the disciplinary divide to incorporate contributions from scientists and other specialists. The Intech committee hopes that its book chapters, journal articles, and other activities will help increase knowledge across disciplines and around the world. To that end the committee invites readers to contribute ideas on how best this objective could be accomplished.

InTech Europe

University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China

Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen