An Adaptive Distributed Computing with Fault Tolerance and Recovery

Mrs.K.Gayathri[#1], Mr.B.Ganesh Prabu[*2]

[#]*Lecturer, Department of Computer Science and Engineering,*
PSNA College of Engineering and Technology, Dindigul,, Tamilnadu.

[1]gayathri1773@gmail.com

[*]*Assistant Professor, Department of Electrical and Electronics Engineering,*
University College of Engineering, Dindigul, Tamilnadu.

[2]ganesh.eee.007@gmail.com

***Abstract*—Today we need high-speed Computers to meet our customer requirements like data warehousing, on-line transaction processing and decision support system solutions. Likewise, the world of science & engineering also rely on high performance computing to provide solutions and solve problems. Mainframes, supercomputers and fault-tolerant systems are high cost with limited speed. Distributed systems consisting of independent computers that co-operate as a single system. Distributed computing is sharing of computer resources and services by direct exchange between systems. DCE provides a complete Distributed Computing Environment infrastructure. It provides security services to protect and control access to data, name services that make it easy to find distributed resources, and a highly scalable model for organizing widely scattered users, services, and data. DCE runs on all major computing platforms and is designed to support distributed applications in heterogeneous hardware and software environments. DCE is a key technology in three of today's most important areas of computing. The performance of the DCE is entirely depends on number of participants included in a particular process using multithreading, where number of participants increase, system failure rate is also increased to make the DCE reliable, fault detection, fault tolerance and failure recovery must be done. This Project work aims to design a reliable DCE with the help of Message-based middleware.**

***Keywords*—Distributed computing, Fault tolerance, Failure, Checkpoints**

I. INTRODUCTION

As the number of processors in modern high performance distributed computer systems continues to grow, the issue of fault tolerance is becoming more and more important. Even making generous assumptions on the reliability of a single processor or link, it is clear that as the processor count in high end clusters grows into the hundreds of thousands, the mean-time-to-failure of these clusters will drop from a few years to a few days, or less. The current DOE ASCI computer (IBM Blue Gene L) is designed with 131,000 processors [1]. The mean-time-to-failure of some nodes or links for this system is reported to be only six days on average [1]. In recent years, the trend of the high performance computing has been shifting from the expensive massively parallel computer systems to the clusters of commodity off-the-shelf systems [5]. While the commodity off-the-shelf cluster systems have excellent price performance ratio, there is a growing concern with the fault tolerance issue in such system. The recently emerging computational grids [9] environments have further exacerbated the problem. However, many computational science programs are now designed to run for days or even months. Therefore the mean-time-between-failures (MTBF) of such kind of high performance computing systems are significantly shorter than the running time of many computational science programs. Modern computational science programs need to be able to tolerant the failures. Due to the large process state of such kind of applications, the relatively low I/O bandwidth between memory and the central network disk and the high enough frequency of failures, for these systems, the classical system-level fault tolerance approaches is often either impractical

(an application would spend most of its time taking checkpoints) or infeasible (there is no enough time for an application to save its core to disk before the next failure occurs). Therefore the cheaper application level fault tolerance schemes may be deployed as an alternative in such large computational science programs. However, most application level fault tolerance schemes proposed in literature are non-adaptive in the sense that the fault tolerance schemes incorporated in applications are either designed without incorporating system environments (such as the amount of available memory and the local and network I/O bandwidth, etc) or designed only for a specific system environment. In this paper, we propose a framework under which different fault tolerance schemes can be incorporated in applications using an adaptive method. In our framework, applications will be able to choose the best (minimizing the mean execution time of the application) available fault tolerance schemes at runtime (or dynamically) according to different (or dynamic) system environments[3]. Applications that call this kind of self-adaptive fault tolerant numerical libraries will be able to survive certain processor failures transparently with very low performance overhead. The rest of this paper is organized as follows. Section 2 reviews briefly the existing related literature in checkpointing and rollback recovery. Section 3 explains the motivations of this research. Section 4 presents a self adapting application, level fault tolerance scheme for high performance grid computing. In Section 5, some initial experimental results are presented. Section 6 concludes the paper and discusses future work.

II. FAULT TOLERANCE IN PARALLEL AND DISTRIBUTED SYSTEMS

Fault tolerance techniques can be divided into two big branches and some hybrid techniques. The first branch is Messaging Logging. In this branch, there are three subbranches: Pessimistic Messaging Logging, Optimistic Messaging Logging and Casual Messaging Logging. The second branch is Checkpointing and Rollback recovery. There are also three sub-branches in this branch: Network disk based Checkpointing and rollback recovery, Diskless Checkpointing, and Local Disk based checkpointing. Our research is mainly concentrated on incorporating fault tolerant techniques into tightly coupled large scale high performance computational intensive applications. In the rest of this section, we confine our literature review to checkpointing and rollback recovery schemes instead of general fault tolerance scheme[4]. Most traditional distributed multiprocessor recovery schemes are designed to tolerate arbitrary number of failures. So they store their checkpoint data in a central stable storage. The central stable storage usually has its own fault tolerance techniques to prevent it from failures. But the bandwidth between the processors and the central stable storage is usually very low. Several experimental studies presented in [13] have shown that the main performance overhead of checkpointing is the time spent on writing the checkpoint data to the central stable storage. In [11] and [13], Plank proposed to use diskless checkpointing technique as an approach to tolerant single failures with low performance overhead when stable storage is not available. Diskless checkpointing is a technique where processor redundancy, memory redundancy and failure coverage are traded off so that a checkpointing system can operate in the absence of stable storage. Experimental studies presented in [13, 14] have shown that diskless checkpointing have a much better performance than traditional disk based checkpoint techniques. There are also several papers which compare the performance of different diskless checkpointing schemes. In [4],Chiueh and Deng compare the performance of different diskless checkpointing schemes on a massively parallel SIMD machine. The XOR operation was done following an O(logN) binary tree fashion. The results of their experiment show that the Checkpoint Mirroring is an order of magnitude faster than the Parity Checkpointing, however introduced twice as much memory overhead as Parity Checkpointing. In [14], Silva also did some experimental studies about diskless checkpointing .The experiments were done on an Xplorer Parsytec machine with 8 transputers (T805). Their experimental results show that Checkpoint Mirroring has a much better performance than the n+1 Parity schemes. The drawback is that Checkpoint Mirroring always presents more memory overhead than the n+1 Parity schemes. In [12], Plank also reported that Checkpoint Mirroring has lower performance overhead than Parity checkpointing if the checkpoint data is stored in local disk instead of the memory of the processor. Local disk can also be used to store the checkpoint data. In his paper, coordinated checkpoints are first taken to the local disk of each processor and then Checkpoint Mirroring, n+1 Parity, or Reed-Solomon Coding are used to encode the local checkpoint data to the local disk of other processors. To tolerate arbitrary number of failures with low

performance overhead, Vaidya proposed a two-level distributed recovery approach in [15]. Checkpoint can be done either at the system-level or at the application level. In [14], Silva compared the performance overhead of the system-level checkpointing and the user defined checkpointing. But the degree of the performance improvement is also dependent on specific applications. In summary, a review of the existing fault tolerance research demonstrates that

• To tolerate arbitrary number of failures with low performance overhead, a two-level (or multi-level) recovery scheme should be used.

• If enough memory is available, Checkpoint Mirroring should be used rather than Parity Based Checkpointing.

• If there is no enough memory but there is enough local disk storage available, local disk storage can be used to reduce the checkpoint performance overhead.

• To achieve low performance overhead, user defined checkpointing schemes should be used instead of the system-level checkpointing schemes.

III. MOTIVATIONS FOR SELF ADAPTING FAULT TOLERANCE

From Section 2, we have seen that the previous fault tolerant research works have produced some very precious result. However, there appears to be a significant gap between the fault tolerant research results and their optimal deployment into applications. Each fault tolerance scheme has its own advantages and disadvantages. Different systems have different resource characteristics. What is the best way to incorporate different fault tolerance schemes into applications so that the reliability and survivability is as high as possible while the performance overhead is as low as possible? From the application point of view, it is desirable that fault tolerant high performance applications is able to achieve both high performance and high reliability (survivability) with low fault tolerance overhead no mater under which kind of system environments it is running. To achieve this goal, the best strategy would be to adaptively choose the fault tolerance schemes in applications based on different (or dynamic) system environments that the applications are running. The key idea of our recovery framework is the adaptivity of our checkpoint scheme to different system environments. Our adaptive scheme is similar to Vaidyas two-level recovery scheme in that both schemes take multi-level checkpoint to tolerate arbitrary number of failures with low performance overhead. However Vaidyas recovery technique is static. He consider the availability of the memory and the local disk storage at the software design time, but after the design is finished, the software will never need to check the information of the hardware architecture (such as number of available processors, amount of memory and local disk storage) again. Thus we classify his scheme as static scheme. However, in our scheme, the software will have to check the information of the hardware architecture (such as number of available processors, amount of memory and local disk storage) to decide the optimal checkpoint scheme. Thus, we regard our scheme as adaptive rather than static. The application of this framework to self-adaptive numerical software such as LFC will result in self-adaptive fault tolerant numerical libraries. Applications that use this kind of self-adaptive fault tolerant numerical libraries is able to survive certain processor failures transparently with very low performance overhead.

IV. A SELF ADAPTING APPLICATION LEVEL FAULT TOLERANCE SCHEME

In this section, we present a self adapting application level fault tolerance scheme for high performance grid computing.

A. OVERVIEW

Our goal is to establish a framework under which different fault tolerance schemes can be optimally incorporated in applications using an adaptive method. In our framework, applications will be able to adaptively choose the best (minimizing the mean execution time of the application) available fault tolerance schemes at runtime according to different system environments. Different fault tolerant schemes require different resources. When designing the fault tolerant application, the application developer may not have an a priori knowledge of the system characteristics of the platform where the application will be running on. The system characteristics that is necessary in determining checkpoint schemes may include

• The number of available processors

• The amount of available memory on each processor

• The amount of available local disk storage on each processor

• Whether there is a central fail free stable storage available

• The I/O bandwidth of the local disk storage and the central stable storage of each processor
• The network bandwidth between processors
• An estimate of the MTBF of the system environments

Different fault tolerant schemes have different degree of reliability. To tolerate the failure of all processors, a central stable storage is usually necessary. If the main memory is not enough, consider using the local disk. In order to achieve low memory overhead, we also consider Kims checksum and reverse computation method [10]. In order to achieve transparency, consider and incorporate the fault tolerance in numerical libraries such as LFC[12]. Because we are using an application level
approach, it is also possible to consider the characteristics of the application.

B. A MULTI-LEVEL SELF ADAPTIVE RECOVERY SCHEME

Assume a processor can access the following five types of storage in the computing system
• local memory of the processor
• local disk of the processor
• neighbor processors' memory
• neighbor processors' disk
• central stable storage

If one type of storage is not available in the system, then we assume there are zero bytes of that type of storage in the system. Assume a node failure also means that both its memory and its local disk becomes unavailable. Which kind of checkpoint schemes (or combination, or modification of schemes) is best for a specific system is affected by many factors. At the present time, we only consider the following factors:
• The amount of available storage of each kind
• The overhead of each checkpoint scheme (which is mainly dependent on the bandwidth of each storage and the characteristics of that checkpoint schemes)
• The failure distribution of the system.
• The characteristics of the application
• The number of available processors for this application.

Just as shown in existing research works, on most systems, the performance of these five basic recovery schemes is increasing (but it is also possible in the future to perform experiments to decide the performance of different schemes at run time). Since we also know the degree of fault tolerance of each basic scheme, which combination to be chose is mainly dependent on the availability and the amount of each storage. We use this information to choose the combination of the basic recovery scheme. If we can somehow check the MTBF (or the failure rate) of the system in the future, we can use it to decide the checkpoint frequency[10]. Otherwise we decide the checkpoint frequency based on the assumption that the total performance overhead of the fault application does not exceed certain percentage (say 5%). By making decisions at run time, we get the opportunity to know more information about the platform. The application will execute rather than making decisions at the application design time[14].

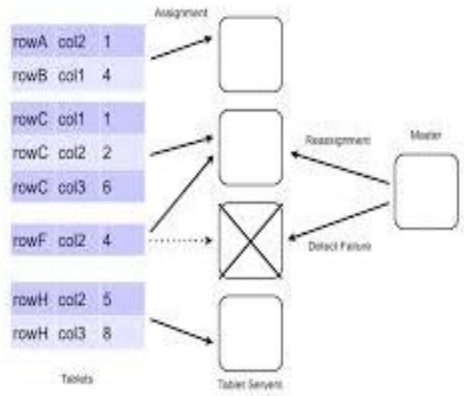

Fig. 1 Automatic Failure Handling

Therefore, we get the opportunity to make better decisions. This is why we can get better performance in a self adapting fault tolerance scheme.

.

V. SNAPSHOT ALGORITHM

Snapshot algorithm whose task is to analyze properties of computations, usually arising from other algorithms. It is, however, surprisingly hard to observe the computations of a distributed system from within the same system. An important building block in the design of algorithms operating on system computations is a procedure for computing and storing a single configuration of this computation called snapshot.

The construction of snapshots is motivated by several applications, of which we list three here. First, properties of the computation, as far as they are reflected within a single configuration, can be analyzed off-line, i.e., by an algorithm that inspects the (fixed) snapshot rather than the (varying) actual process states. These properties include stable properties; a property P of configurations is stable if P(r) r-> =>P (d). If a computation ever reaches a

configuration r for which P holds, P remains true in every configuration d from then on. Consequently, if P is found to be true for a snapshot of the configuration, the truth of P can be concluded for all configurations from then on. Examples of stable properties include termination, deadlock, loss of tokens, and non-reach ability of objects in dynamic memory structure.

Second, a snapshot can be used instead of the initial configuration if the computation must be restarted due to a process failure. To this end, the local state Cp for process P, captured in the snapshot, is restored in that process, after which the operation of the algorithm is continued.

Third, snapshots are a useful tool in debugging distributed programs. An off-line analysis of a configuration taken from an erroneous execution may reveal why a program does not act as expected.

The snapshot algorithm works like this:

1. The observer process (the process taking a snapshot):
    a. Saves its own local state
    b. Sends a snapshot request message bearing a snapshot token to all other processes
2. A process receiving the snapshot token *for the first time* on *any* message:
    a. Sends the observer process its own saved state
    b. Attaches the snapshot token to all subsequent messages (to help propagate the snapshot token)
3. Should a process that has already received the snapshot token receive a message that does not bear the snapshot token, this process will forward that message to the observer process. This message was obviously sent before the snapshot "cut off" (as it does not bear a snapshot token and thus must have come from before the snapshot token was sent out) and needs to be included in the snapshot.

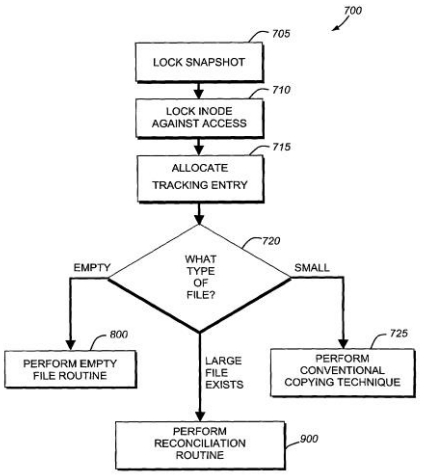

Fig. 2 Snapshot Algorithm Process

From this, the observer builds up a complete snapshot: a saved state for each process and all messages "in the ether" are saved.

VI. REASONS FOR USING FAULT-TOLERANT ALORITHMS

Increasing the number of components in a distributed system means increasing the probability that some of these components will be subject to failure during the execution of a distributed algorithm. Computers in a network may fail, processes in a System can be erroneously killed by switching off a workstation, or a machine may produce an incorrect result due to, memory malfunctioning. Modern computers are becoming more and more in any individual computer. Nonetheless, the chance of a failure occurring at some place in a distributed system may grow arbitrarily large when the algorithm each time a failure occurs, algorithms should be designed so as to deal properly with such failures.

Vulnerability to failures is also a concern in sequential computations, in safety-critical application, or if a computation runs for a long time and produces a non-verifiable result. Internal checks protect against errors of some types but of course no protection can be achieved against the complete loss of the program or erroneous changes in its code. Therefore the possibilities of fault-tolerant computing by sequential algorithms and uni-processor computing systems are limited. In stabilizing algorithms correct processes can be affected by failures, but the algorithm is guaranteed to recover from any arbitrary configuration when the processes resume correct behavior.

VII. FAILURE MODELS

To determine how the correctly operating processes can protect themselves against failed processes, assumptions must be made about how a process might fail. In the following chapters it is always assumed that only processes can fail; channels are reliable. Thus, if a correct process sends a message to another correct process, receipt of the message within finite time is guaranteed.(A failing channel can be modeled by a failure in one of the incident processes, for example , an omission failure.) As an additional assumption, we always assume that each process can send to each other process.

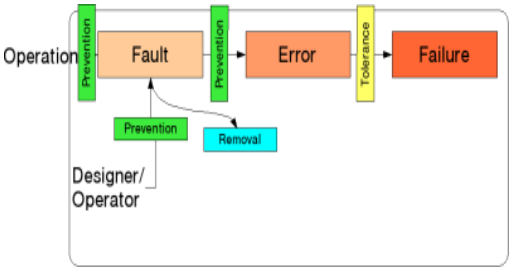

Fig. 3 Failure Models

The fault models are:

1)Initially dead processes: A process is called initially dead if it does not execute a single step of its local algorithm.

2)Crash model: A process is said to crash if it executes its local algorithm correctly up to some moment, and does not execute any step thereafter.

3)Byzantine behavior: A process is said to be Byzantine if it executes arbitrary steps that are not in accordance with its local algorithm. N particular, a Byzantine process may send messages with an arbitrary content.

VIII. FAILURE DETECTION

The impossibility of solving consensus in asynchronous systems has led to weaker problem formulations and stronger models.Failure detectors are new widely recognized as an alternative way to strengthen the computation model. Studying synchronous models is practically motivated because most distributed programming environments do provide clocks and timers in some way. With failure detectors, the situation is similar; quite often the run-time support system will return error messages upon an attempt to communicate with a crashed process. However, these error messages are not always absolutely reliable. It is therefore useful to study how reliable they must be to allow a solution for the consensus problem.

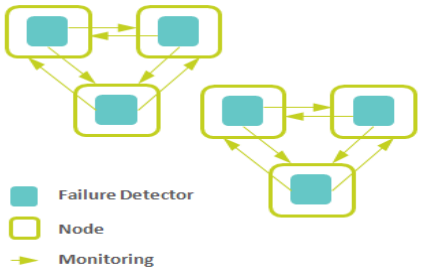

Fig. 4 Failure Detection

IX. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed self adapting fault tolerance scheme experimentally.

The application we used to perform experiment is the PCG code described in [2]. The number of simultaneous processor failures we want to survive is one. The total number of processors we used in PCG is sixteen. The programming environment we used is FT-MPI [6, 7, 8]. All experiments were performed on a cluster of 32 Pentium IV Xeon 2.4 GHz dual-processor nodes. Each node of the cluster has 2 GB of memory and runs the Linux operating system. The nodes are connected with a Gigabit Ethernet. The timer we used in all measurements is MPI Wtime. Table 1 reports the time for performing one checkpoint for both the SSAC and the NMPCschemes. By changing the input problem size in PCG, we varied the amount of data that need to be checkpointed from 100 MBytes to 1,000 MBytes. The results in Table 1 indicate that the SSAC scheme performs better than the NMPC scheme when the size of checkpoint is less than 500 MBytes. However, when the size of checkpoint is larger than 500 MBytes, the SSAC scheme performs approximately the same as the NMPC scheme. Therefore, the use the Neighbor Memorybased Checkpoint Mirroring scheme (which has lower performance overhead but high memory overhead than NMPC) is recommended.

X. CONCLUSION AND FUTUREWORK

Mainframes and Parallel computers are highly reliable and cost number crunches. The recent decade of client server technology evolution indicates cost as a prime factor. There by , identifying distributed object based solutions and

linux clusters.With full advantage of economy distributed object based solution still stays as a bumbling amateur. As the first pedestal, we had designed a Reliable Distributed Computing Environment in windows platform. The project was listed with a cry problem of Encryption. It was successful and has paved hope towards a lot of future scope.

Current implementation of the distributed computing environment is designed using Message based Middleware this can be enhanced with Naming Service. Current implementation of the distributed computing environment is designed to run in Microsoft Network under windows-NT architecture (IntraNet), this can be extended to Internet with the help of WEB SERVERS. Intelligence can be added to enhance the job distribution.

REFERENCES

[1] N. R. Adiga and et al. An overview of the BlueGene/L supercomputer. In Proceedings of the Supercomputing Conference (SC'2002), Baltimore MD, USA, pages 1– 22, 2002.

[2] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault tolerant high performance computing by a coding approach. In Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 14-17, 2005, Chicago, IL, USA. ACM, 2005.

[3] Z. Chen, J. Dongarra, P. Luszczek, and K. Roche. Self-adapting software for numerical linear algebra and LAPACK for clusters. Parallel Computing, 29(11- 12):1723–1743, November-December 2003.

[4] T. Chiueh and P. Deng. Evaluation of checkpoint mechanisms for massively parallel machines. In FTCS, pages 370–379, 1996.

[5] J. Dongarra, H. Meuer, and E. Strohmaier. TOP500 Supercomputer Sites, 28th edition. In Proceedings of the Supercomputing Conference (SC'2006), Pittsburgh PA, USA. ACM, 2006.

[6] G. E. Fagg and J. Dongarra. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world. In PVM/MPI 2000, pages 346–353, 2000.

[7] G. E. Fagg, E. Gabriel, G. Bosilca, T. Angskun, Z. Chen, J. Pjesivac-Grbovic, K. London, and J. J. Dongarra. Extending the MPI specification for process fault tolerance on high performance computing systems. In Proceedings of the International Supercomputer Conference, Heidelberg, Germany, 2004.

[8] G. E. Fagg, E. Gabriel, Z. Chen, , T. Angskun, G. Bosilca, J. Pjesivac-Grbovic, and J. J. Dongarra. Process fault-tolerance: Semantics, design and applications for high performance computing. Submitted to International Journal of High Performance Computing Applications, 2004.

[9] I. Foster and C. Kesselman. The Grid: Blueprint for a New Computing Infrastructure. Morgan Kauffman, San Francisco, 1999.

[10] Y. Kim. Fault Tolerant Matrix Operations for Parallel and Distributed Systems. Ph.D. dissertation, University of Tennessee, Knoxville, June 1996.

[11] J. S. Plank and K. Li. Faster checkpointing with n+1 parity. In FTCS, pages 288–297, 1994.

[12] J. S. Plank. Improving the Performance of Coordinated Checkpointers on Networks of Workstations using RAID Techniques. In 15th Symposium on Reliable Distributed Systems, pages 76–85, 1996.

[13] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst., 9(10):972–986, 1998.

[14] L. M. Silva and J. G. Silva. An experimental study about diskless checkpointing. In EUROMICRO'98, pages 395–402, 1998. [15] N. H. Vaidya. A case for two-level recovery schemes. IEEE Trans. Computers, 47(6):656–666, 1998.

[15]. Qiming Chen ; Meichun Hsu ; Castellanos, M. Software Engineering, Artificial Intelligence,Networking andParallel/Distributed Computing (SNPD), 2013 14th ACIS International Conference on Digital Object Identifier: 10.1109/SNPD.2013.36 Publication Year: 2013 , Page(s): 261 – 266, IEEE conference publications.

[16]. Han, Y.S. ; Hung-Ta Pai ; Rong Zheng ; Wai Ho Mow Communications, IEEE Transactions on Volume: 62 , Issue: 2 Digital Object Identifier: 10.1109/ TCOMM.2013.122313.130492 Publication Year: 2014 , Page(s): 385 – 397. IEEE journals & magazines.

[17]. Pezoa, J.E. ; Hayat, M.M. Parallel and Distributed Systems, IEEE Transactions on Volume: 25 , Issue: 4 Digital Object Identifier: 10.1109/ TPDS.2013.78 Publication Year: 2014 , Page(s): 1034 – 1043 IEEE journals & magazines.

