Fault Tolerance for Conjugate Gradient Solver Based on FT-MPI

Weizhe ZHANG[1], Hui HE[2]

Harbin Institution of Technology,
92 West Dazhi, Harbin, 150001, China,
wzzhang@hit.edu.cn, hehui@hit.edu.cn

Abstract: Grid computing is characterized by high speed, large scale, large task quantity, and long cycles. Such characteristics prevent the waste of large amounts of computing power and time that can be attributed to system errors. Moreover, such features provide the fault tolerance of computing resource nodes in the structural system of grid computing, which has become a key issue in the field. This paper describes the current fault-tolerant message passing interface library, designs a grid computing-based task migration and recovery model, and then identifies the functional architecture of each module of the mode. Further analysis and comparison were conducted on the storage mechanism of the fault-tolerant checkpoint of the model as well as its information-encoding algorithm. Finally, the realization of a Checksum algorithm-based fault-tolerant conjugate gradient solver shows the validity of the theory.

Keywords: fault tolerance; CG solver; FT-MPI; computational grid.

1. Introduction

With the rising expansion and continuous improvement in the complexity of the grid-computing scale, the probability of system component failure, including software and hardware, continues to increase. Moreover, the grid-computing application executes a large volume of tasks that have long life cycles. In the case of non-fault-tolerant measures, system errors cause processes with long life cycles to restart, which consequently degrades the calculations made before the error occurred. Additionally, other error-free processes that are related to the erroneous process may have to restart to return the entire system to its original state, thus causing enormous losses. In some cases, a compromise between long periods of calculation and stable system operation remains a challenge.

Thus, studying the fault tolerance of a grid-computing system is of great significance. Designing an effective fault-tolerant mechanism can effectively prevent data loss. Such a mechanism can also prevent process from restarting after an error occurs. Thus, an effective fault-tolerant mechanism will aid in the construction of a reliable, consistent, continuous, low-cost, and high-end grid-computing system.

Linear equations are widely used in scientific and engineering computations. Such equations have long computing cycles and require large amounts of calculations, which can be solved by using a grid-computing platform. The conjugate gradient (CG) method [13, 16-18], also known as the conjugate vector method, is an iterative method for solving linear equations. This method is characterized by the capability to provide an approximate solution for high-order equations. Such an approximate solution meets accuracy requirements only when the number of iterations is considerably smaller than the order. The fault-tolerance of the CG solver in a grid -computing platform is of great practical value.

This paper primarily investigates fault tolerance based on the fault-tolerant message passing interface (FT-MPI) [2, 3] library and then uses checkpoint technology to realize the CG method. When failure occurs in a node of the grid-computing system during calculation, a checkpoint is reasonably set and checkpoint information is effectively encoded for the backup and migration of computing tasks. This process prevents the failure of a single node from causing the entire system to stop operating normally.

The remainder of this paper is organized as follows: Section 2 describes related studies on the existing FT-MPI library. Section 3 provides a task migration and recovery model for a grid-computing environment. Section 4 presents a Checksum checkpoint-based fault-tolerant CG solver, based on the given model and then discusses the related experimental results. Finally, the conclusion and plans for future work are presented.

2. Related Works

The international community has adopted different fault-tolerant mechanisms in studies of FT-MPI to achieve fault tolerance in grid

computing. Such mechanisms include Co-Check MPI [4], Starfish MPI [5], MPICH-V [6], LA-MPI [7], MPI/FT [8], MPI-FT [9], and FT-MPI. A number of studies have achieved FT-MPI, but each MPI has its own advantages and disadvantages, and no standard can be uniformly accepted. FT-MPI is the more common among these mechanisms. Thus, we use FT-MPI as a platform to achieve a fault-tolerant grid-computing system.

Co-Check MPI was developed by the Technical University of Munich and is the first implementation of FT-MPI. This mechanism adopts check-point/roll back technology and is a complete application for the achievement of MPI fault tolerance based on the checkpoint of the Condor library.

Starfish MPI was developed by the University of Illinois. This application can adapt to dynamic change and has a built-in checkpoint function. Starfish MPI combines strict atomic group communication technology and checkpoints/restart agreement as well as the support cooperation and non-cooperation checkpoint mechanism.

MPICH-V was developed at the Innovative Computing Laboratory, University of Tennessee, Knoxville and uses different communication protocols (including Myrinet) to support fault-tolerant function. MPICH-V establishes distributed and non-cooperative checkpoints and takes advantage of the dynamic characteristics of checkpoint technology; for example, nodes can join and leave the grid environment at any time.

LA-MPI originated from the Los Alamos National Laboratories and served to provide a reliable channel for messages between processes, but not to control process failure. To achieve this objective, the communication layer is divided into two parts: the memory and message management layer and the sending and receiving layer.

MPI/FT was developed by the Columbia University through the introduction of central coordination and of a copy of MPI process mechanisms to conduct fault tolerance.

MPI-FT was developed by Paraskevas Evripidou in Cyprus University. This mechanism supports a number of applications under the master-slave mode and enables all communication nodes constructed in the grid to include free processes. Once an error occurs, the free processes conduct initialization.

3. Task Recovery Model

3.1 Model

In this model, a grid-computing task has *n* computing nodes, *m* backup nodes, *k* idle nodes, a master node, and a backup node of the master node, as shown in Figure 1.

The functions of each node in fault-tolerant applications for grid computing are as follows:

Master node: (1) stores a list of information regarding backup nodes including the number, identification, and the information storage capacity of nodes; regularly updates the list of information; and regularly monitors the condition of the backup nodes. (2) stores information on the idle node, including name, communication time, and load condition; sends regular queries to idle nodes for updates. (3) sends certain global information to the nodes, the backup and idle nodes; for example, the information table on idle nodes. (4) performs task scheduling in the master-slave mode.

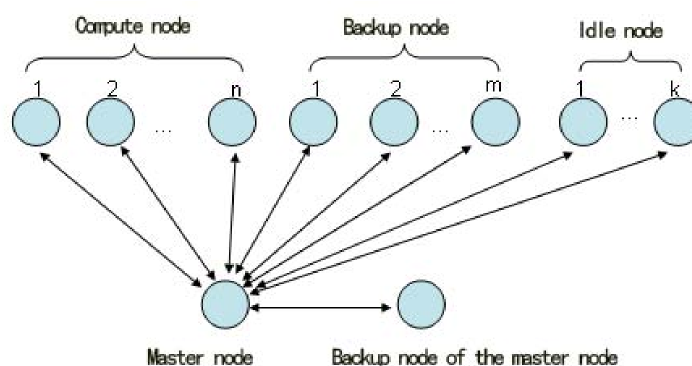

Figure 1. Grid computing task migration and recovery model

Backup node of the master node: (1) serves as real-time backup for master node data. (2) monitors the operational status of the master node. The backup node takes over the task the master node when the latter fails.

Compute node: (1) performs computing application tasks. (2) establishes checkpoint information. (3) recovers the data and computing tasks of the node.

Backup node: (1) calculates and updates the verification information of the checkpoint. (2) recovers the data and computing tasks of the failed node.

Idle node: can be used as a replacement for failed compute or backup nodes.

3.2 Checkpoint storage and information encoding algorithm

The checkpoint is a fault-tolerant technology that is simple, efficient, and practical and has thus been widely applied in FT-MPI. The functional structure of the above model shows that a checkpoint is capable of saving data and restoring the status of the running program and is thus essential for the grid-computing system to achieve stable and effective fault tolerance.

Checkpoint storage. Checkpoint storage is classified into two types: file checkpoint and diskless checkpoint [10].

(1) File storage checkpoints. Storage is usually achieved by saving checkpoint information onto a reliable media (such as a disk or disk array) as a file. The advantage of such checkpoints is high reliability provided that a local or backup checkpoint can be saved onto a relatively stable storage medium. The disadvantages are as follows:

The time involved in setting up and in recovering a checkpoint is relatively long because reading and writing from the external medium is relatively slow.

If file pointer information is not saved by the checkpoint, then the error would not be recovered when checkpoint setup fails.

(2) Diskless checkpoint. Diskless checkpoint is a technology that is capable of storing a large-scale computing state onto a distributed system; it does not depend on a stable storage device. Under such mechanism, the state of each computing process is saved in the memory of the local region. In addition, the checkpoint codes are stored in a memory that is unrelated to computing applications of the local area.

When an error occurs, the state of each active process reverts to the previous checkpoint state; the state of the failed checkpoint can be recovered from processes that did not fail and from those that stored the checkpoint codes. Through redundancy in its memory, a diskless checkpoint eliminates the resource cost of the checkpoint mechanism in a common distributed system.

The advantages of a diskless checkpoint are as follows:

Storage is more convenient because only a few storage variables or data structures must be set up.

Checkpoint setup and error recovery run at high speeds. The speed of memory reading and writing is faster compared with reading and writing through file storage.

When failure occurs during checkpoint setup, the program can still recover properly based on the checkpoint.

The disadvantages are as follows:

Storing large amounts of checkpoint information is impossible. The memory capacity of a diskless checkpoint is significantly smaller than that of disk media, which certainly affects computations that require a large amount of memory.

Reliability is relatively lower than that of a file checkpoint. The loss of all memory data is possible in case of a sudden power failure.

Checkpoint information-encoding algorithm. Encoding a reasonable amount of checkpoint information not only saves storage space, but also increases the efficiency and of fault tolerance of the system. The most commonly used encoding algorithms are the Checksum [11] and Reed–Solomon algorithms [12].

(1) Checksum algorithm

Checksum is a simple, efficient, and useful information-encoding algorithm. Checksum can effectively encode checkpoint information from multi-memory computing, thus achieving system redundancy fault-tolerance.

Encoding is relatively simple when applying the checksum algorithm. Thus, less additional time is required in computing the coding value, making the efficiency of system recovery

Table 1. Comparison with Checksum and Reed-Solomon algorithm

Encoding Algorithm	**Checksum**	**Reed-Solomon**
Fault Tolerance	Tolerate multiple, single-node failure	Tolerate at most m nodes failure at the same time several times
Space Overhead	Need a check node	Need m check nodes
Time Overhead	Less computing time, Less communication time	More computing time More communication time
Difficulty of Achieving	Relatively simple	Relatively difficult
Speed of Recovery	Faster	Slower

relatively high. However, this method can only tolerate failure in a single node but not in multiple nodes.

(2) Reed–Solomon algorithm

The Reed–Solomon algorithm is an information-coding algorithm that is applied to multi-memory-distributed data. This algorithm can realize redundancy fault-tolerance in various failed equipment. Below is a complete proof for the Reed–Solomon algorithm's fault-tolerant feasibility.

Definition: Write the function F_i (i=1, 2… m) as a linear combination of the data:

$$c_i = F_i\left(d_1, d_2, \ldots d_n\right) = \sum_{j=1}^{n} f_{i,j} d_j \ , \tag{1}$$

That is:

$$\begin{cases} f_{1,1}d_1 + \ldots + f_{1,n}d_n = c_1 \\ \quad \vdots \qquad \vdots \\ f_{m,1}d_1 + \ldots + f_{m,n}d_n = c_m \end{cases} , \tag{2}$$

Note that $F = \left(f_{i,j}\right)_{m \times n}$, referred to as Check Matrix.

Theorem: When the number of failed equipment k (k <= m) and any arbitrary sub-matrix in the check matrix are both non-singular, the entire system can be successfully resumed by using the Reed–Solomon algorithm.

Proof: Suppose k_1 data device and k_2 calibration device errors, which indicates that k_1 data words and k_2 checking words are unknown. Moreover, $k = k_1 + k_2$. Equation Group (2) then becomes an equation group with m equations and k unknowns.

If $k > m$, then the number of unknown quantities exceeds the number of equations. Thus, Equation Group (2) has no unique solution, and the missing data in the failed equipment cannot be restored. Thus, when the number of failed equipment exceeds the number of calibration equipment, the system cannot be restored.

If $k \le m$, then the number of unknowns is less than or equal to the number of equations. Through the appropriate selection of the check matrix F, the equation can have a unique solution, and the missing data in the failed equipment can be restored. Thus, when the number of failed equipment is less than or equal to the number of calibration equipment, the system can be restored through the appropriate selection of the check matrix F.

Without loss of generality, we suppose that data devices $D_{j_1}, D_{j_2}, \ldots, D_{j_{k_1}}$ and checking devices $C_{i_1}, C_{i_2}, \ldots, C_{i_{k_2}}$ fail, which means that data words $d_{j_1}, d_{j_2}, \ldots, d_{j_{k_1}}$ and checking words $c_{i_1}, c_{i_2}, \ldots, c_{i_{k_2}}$ become unknown. Equation Group (2) then becomes

$$\begin{cases} f_{i_{k_2+1},j_1} d_{j_1} + \ldots + f_{i_{k_2+1},j_{k_1}} d_{j_{k_1}} = c_{i_{k_2+1}} - \sum_{t=k_1+1}^{n} f_{i_{k_2+1},j_t} d_{j_t} \\ \qquad \vdots \\ f_{i_m,j_1} d_{j_1} + \ldots + f_{i_m,j_{k_1}} d_{k_1} = c_{i_m} - \sum_{t=k_1+1}^{n} f_{i_m,j_t} d_{j_t} \end{cases} \tag{3}$$

$$\begin{cases} c_{i_1} = f_{i_{i_1},1} d_1 + \ldots + f_{i_1,n} d_n \\ \qquad \vdots \\ c_{i_{k_2}} = f_{i_{k_2},1} d_1 + \ldots + f_{i_{k_2},1} d_n \end{cases} \tag{4}$$

Notably, correlation coefficient matrix is F_r for Linear Equation Group (3). If F_r is a full column rank, then d_{j_t} $\left(t = 1, 2\ldots, k_1\right)$ can be restored through Linear Equation Group (3),

Table 2. Result of fault-tolerant CG operate normally

Number of processes	Number of iterations	Number of checkpoints	Normal computation time	Normal Communication time	Total time
3	7825	40	0.536822	8.974775	9.619499
4	7673	39	0.45834	13.2111	13.77294
5	7832	40	0.485311	19.7759	20.36834
6	7807	40	0.466285	21.21815	21.79224
7	7605	39	0.433963	21.08805	21.62652

which can facilitate the restoration of $C_{i_t}\ (t = 1, 2, \ldots, k_2)$ through Linear Equation Group (4). Finally, the entire system can be restored.

Therefore, the restoration of missing data on the failed node is based on whether the matrix F_r is a full-column rank. To enable the failed equipment to be restored, F_r can be any sub-array of matrix *F*. Likewise, given that any sub-array of the matrix *F* is nonsingular and that the number of storage device failures is k ($k \leq m$), F_r is full-column rank. We can see that when any sub-array of the check matrix *F* is nonsingular with k ($k \leq m$) storage device failures, the entire system can be restored.

4. Implementation

The fault-tolerant CG [13] solver comprises three modules: the initialization, calculation, and recovery modules.

(1) Initialization module

This module initializes the CG solver's computing environment, which includes establishing the computing process and the backup process, obtaining matrix A, initializing each calculation variable, and so on.

The steps are as follows:

Step 1: The process, which may be a normal or a restarting process, is identified. If the process restarts, it is marked as a restarting process and then the recovery of computing data is initialized. However, if the process is normal, only the computing environment is initialized.

Step 2: A computing workspace, comm_work, is created.

check_proc = n;

MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

MPI_Group_excl(orig_group, 1, &check_proc, &group_work);

MPI_Comm_create(MPI_COMM_WORLD, group_work, &comm_work);

Step 3: Matrix A is obtained. Matrix A is an $m \times m$ sparse matrix that is stored in a file in the Harwell–Boeing format [14, 15, 19, 20].

Step 4: Vector b is initialized. We set $b = A \times L$ (L is a given vector) to facilitate result verification. Finally, the result X=L is obtained through the CG solver.

Step 5: The vector space is applied.

All manuscripts must be **MsWord®** Documents and must comply with publisher's instructions. The paper should be in English and should have the following structure:

(2) Calculation Module

This module completes iteration computing. The Checkpoint function is specifically intended for the completion of the checkpoint setup process. During the calculation process, the system monitors each MPI function call. When a function call is found wrong, the module marks the process state as "Need to restore" and then instantly converts into the Recovery Module.

(3) Recovery Module

This module calculates the recovery process. First, the module calls the MPI statement to restart a process automatically to replace the failed process. Restarting the process is required to restore the computing environment and the calculating data. A normal process needs to restore its state to the checkpoint state.

5. Experiments

Initial conditions: The checkpoint step = 1 checkpoint/200 iterations and calculation accuracy error = 1.0e-30. A process is arbitrarily killed to simulate a single-node failure situation and to test the recovery time.

5.1 Normal execution

We allowed the fault-tolerant CG Solver to operate normally in FT-MPI to test the time parameters. The test prerequisite is as follows: the checkpoint step should create one checkpoint every 200 iterations. The results are given below.

Figure 2 shows a comparison among the computation, communication, and total times. Notably, the number of processes in Figure 2 is the same as the number of calculation processes, excluding the backup process.

Figure 2 shows that the difference in calculation time between fault-tolerant CG and non-fault-tolerant CG is very small, and such difference can be attributed to the network status and machine status. For example, when the number of processes is three, the calculation time of fault-tolerant CG is less than that of non-fault-tolerant CG. This difference evidently results from the network and the machine status. However, when outside conditions are constant, the calculation time of fault-tolerant CG is definitely longer than that of non-fault tolerant CG because of the time required in setting up the checkpoint.

Both communication times in Figure 2 are almost the same as those in Figure 3. Except for several singular points, the additional time cost of other fault-tolerant CGs is lower, and the results are satisfactory.

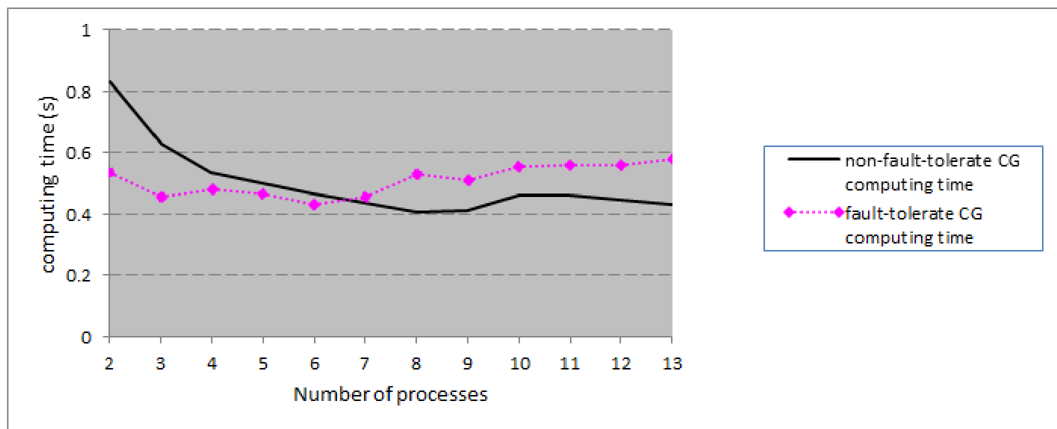

Figure 2. Comparison of computing time between fault-tolerant and non-fault-tolerant CG

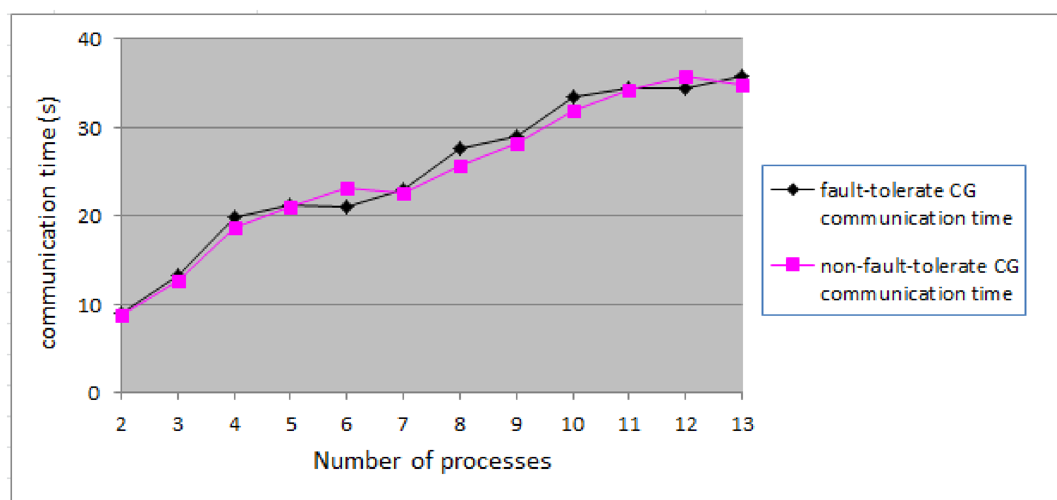

Figure 3. Comparison of communication time between fault-tolerant and non-fault-tolerant CG

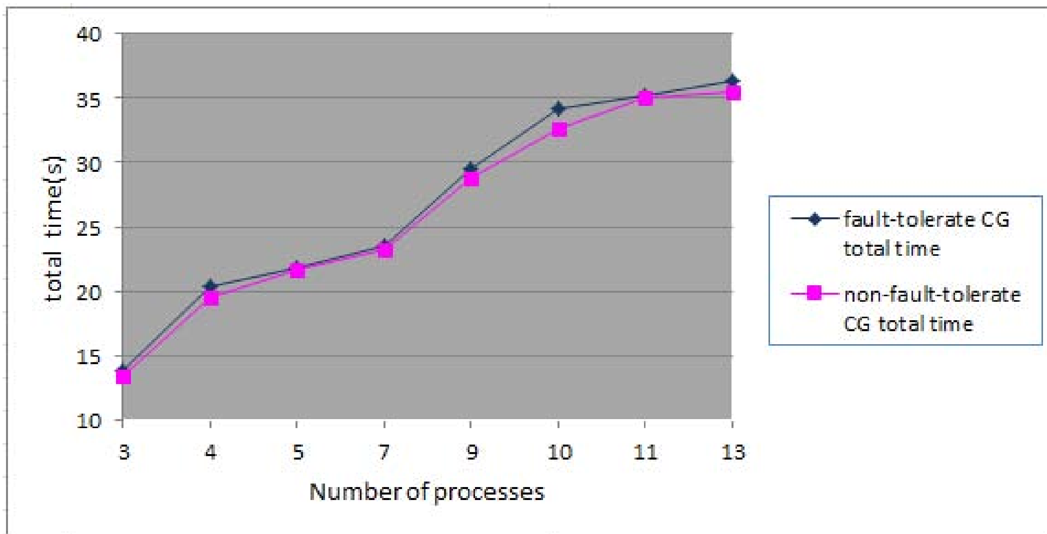

Figure 4. Comparison of total time between fault-tolerant and non-fault-tolerant CG

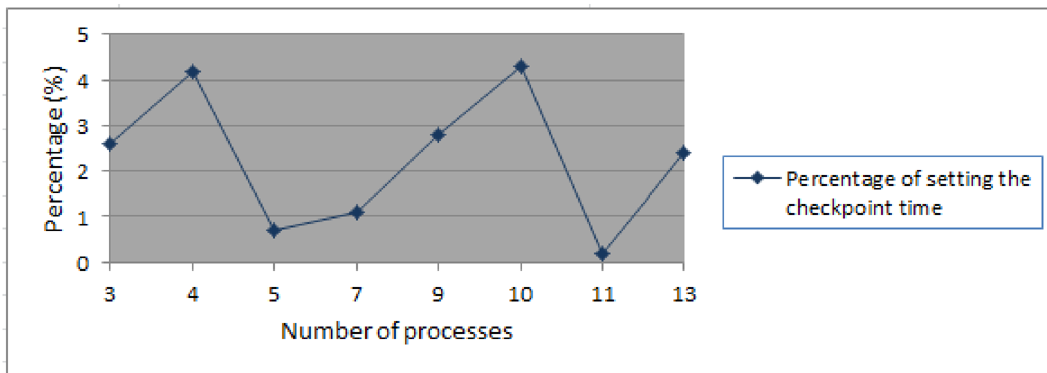

Figure 5. Percentage of checkpoint setup time

Figure 4 indicates no remarkable difference in total time between fault-tolerant and non-fault-tolerant CG. Figure 5 illustrates the cost of setting checkpoints as rather low, at no more than 5% of the total time. In fact, checkpoint setup accounts for only 2.29% of total time on average. Such a result is calculated without special points and is obtained through the network conditions and machine workload. However, given that the network conditions and machine workload have already been included in the points used for computation, the cost should be less than 2.29% of total time. Considering these special points, by simply adding and averaging, we can obtain the ratio of the time required for checkpoint setup against total time, which is 1.61%. Compared with the costs involved in restarting and recounting, this cost is very low.

Table 3. Testing result of the different checkpoints step when the number of processes is 8

Number of processes	Checkpoint step length of (Iteration times)	Number of checkpoints	Computing time(s)	Communication time (s)	Total time(s)
8	50	151	0.652597	23.55539	24.32476
	100	76	0.502503	22.90244	23.50767
	200	38	0.457846	22.89871	23.46044
	400	19	0.535683	22.64116	23.30269
	600	13	0.515328	22.84472	23.48475

5.2 Comparison among various checkpoint steps

We then test each time parameter under various checkpoint steps when the number of processes is eight. Table 3 shows that as the number of checkpoints increases, the total time also increases, albeit in small increments. The desired number of checkpoints is related to the actual amount of computation. The experiments reveal the optimal checkpoint step to be 76 for fault-tolerant CG solvers, which indicates one checkpoint for every 100 iterations.

5.3 Comparison among different checkpoint storage methods

The storage method can be either in-memory storage or file-storage. In Figure 6, we ran a test on file-storage checkpoints and in-memory storage checkpoints. The test shows that file-storage checkpoints have a higher time cost than in-memory storage. This finding is reasonable considering that the read/write speed of files for file-storage is significantly slower than that for in-memory storage.

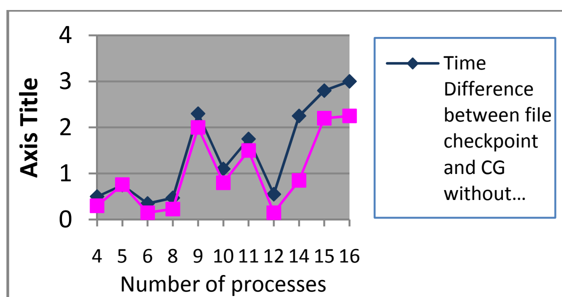

Figure 6. Comparison of file-storage checkpoints and in-memory storage checkpoints

5.4 Fault-tolerance test

Table 4 shows that during the migration and restoration of computing tasks, restructuring in the communication domain has consumed most of the time, whereas the restoration of computing workspaces and data only consumed a small amount of time. Both restorations account for 7.89% of total time. By simply adding and averaging each percentage data, we can obtain the ratio of single restoration time against total fault-tolerance CG execution time as 5.41%.

5.5 Test summary

This chapter provides the detailed testing process for the computing, communication time, and total execution times of fault-tolerant CG solvers in normal execution. From these tests, the efficiency of establishing checkpoints can be retrieved, that is, establishing checkpoints requires 1.61% of the total execution time. Further, various checkpoint steps are tested, and the optimal checkpoint step for fault-tolerant CG solvers is calculated. Tests comparing the parameters of in-memory storage checkpoints and file-storage checkpoints indicate that the latter require more time than the former. Finally, we tested the fault tolerance of fault-tolerant CGs, that is, the migration and restoration of computing tasks when a single process fails, under various numbers of checkpoints. We found that time costs mainly come from the copying task in a communication domain. We also proved that fault-tolerant CGs can tolerate any number of failures in a single process and that one restoration involves approximately 5.41% of total execution time.

5. Conclusion

In this article, we discussed the process for the migration and restoration of multi-storage computing tasks by using a checkpoint algorithm based on the FT-MPI library for implementation in fault-tolerant grid computing. We introduced the current FT-MPI library. We likewise presented a task migration and restoration model based on grid computing, wherein we analyzed and compared the checkpoint storage method and checkpoint information encoding algorithm. Furthermore, we presented a complete proof on the fault-tolerance feasibility of the Reed–Solomon algorithm. Finally, a fault-tolerant CG solver was implemented based on the principle of this model. Test results prove that the fault-tolerant CG solver is redundancy fault-tolerant in single node failure for as many times as possible.

Given that the checkpoint algorithm used for this fault-tolerant CG solver is Checksum, only tolerate the failure of one node can be tolerated at any given time. When multiple nodes fail simultaneously, the algorithm cannot implement fault-tolerance for the system. However, the use of the Reed–Solomon algorithm can mitigate this problem. Therefore, the next step is to implement fault-tolerant CG solvers based on the Reed–Solomon algorithm and then analyze and compare the results with that when the checksum algorithm is used.

Table 4. Check-points test in fault-tolerance CG solvers

Number of processes	Time of copying the communication domain(s)	Time of restoring workspace(s)	Time of restoring data(s)	Total time of single restoration(s)	Normal total time (s)	The percentage of restoration time (%)
3	0.984357	0.0327155	0.046062	1.0631345	9.619499	11.05
4	1.048805667	0.02968025	0.052247	1.13073292	13.77294	8.21
5	1.12095425	0.0215512	0.091045	1.23355045	20.36834	6.06
6	1.1745138	0.021662917	0.072278	1.26845472	21.79224	5.82
7	1.239471333	0.018257929	0.094061	1.35179026	21.62652	6.25
8	1.275007857	0.016904188	0.077942	1.36985405	23.46044	5.84
9	1.335297438	0.015148167	0.082352	1.4327971	28.32543	5.06
10	1.389583444	0.01621565	0.082614	1.48841309	29.538541	5.04
11	1.43583375	0.014203136	0.086411	1.53644789	34.137064	4.50
12	1.509873	0.013151333	0.094145	1.61716933	35.191527	4.60
13	1.571477875	0.014109846	0.097989	1.68357672	35.134292	4.79
14	1.631156	0.013227571	0.130361	1.77474457	36.407385	4.87
15	1.663123893	0.0137886	0.112573	1.78948549	38.639928	4.63
16	1.6909015	0.013524719	0.132652	1.83707772	39.647513	4.63

In addition, the checkpoint steps and the failure characteristics of each node in the system directly affect the execution efficiency of fault-tolerant systems. Hence, we will conduct further research on the relationship model of checkpoint set mechanisms as well as implement and study actual algorithms to improve the efficiency of fault-tolerant systems.

Acknowledgements

Weizhe Zhang is supported in part by the National Grand Basic Research Program (973 Program) of China under grant No. 2011CB302605, National Natural Science Foundation of China (NSFC) under grant No. 61173145.

REFERENCES

1. FOSTER, I., C. KESSELMAN, **The Grid: Blueprint for a New Computing Infrastructure**. Morgan Kaufmann, San Fransisco, CA. 1999 http://mkp.com/grids.
2. FAGG, G. E., A. BUKOVSKY, J. J. DONGARRA. **Harness and Fault Tolerant MPI**. Parallel Computing, 2001.
3. FAGG, G. E., E. GABRIEL, G. BOSILCA, T. ANGSKUN, Z. CHEN, J. PJESIVAC-GRBOVIC, K. LONDON, J. J. DONGARRA. **Extending the MPI Specification for Process Fault Tolerance on High Performance Computing Systems**. University of Tennessee, Knoxville, USA, 2004.
4. STELLNER, G., **Cocheck: Checkpointing and Process Migration for MPI**. In Proceedings of the 10th International Parallel Processing Symposium (IPPS ’96), Honolulu, Hawaii, 1996.
5. AGBARIA, A., R. FRIEDMAN, **STARFISH**: **Fault-tolerant Dynamic MPI Programs on Clusters of Workstations**. In 8th IEEE International Symposium on High Performance Distributed Computing, 1999.

6. BOSILCA, G., A. BOUTEILLER, F. CAPPELLO, S. DJILALI, G. F´EDAK, C. GERMAIN, T. H´ERAULT, P. LEMARINIER, O. LODYGENSKY, F. MAGNIETTE, V. N´ERI, A. SELIKHOV. **MPICH-V**: **Toward a Scalable Fault Tolerant MPI for Volatile Nodes**. In Super-Computing, Baltimore USA, Novembre 2002.
7. GRAHAM, R. L., S.-E. CHOI, D. J. DANIEL, N. N. DESAI, R. G. MINNICH, C. E. RASMUSSEN, L. D. RISINGER, M. W. SUKALSKI. **A network-failure-tolerant message-passing system for terascale clusters**. In ICS. New York, USA, June. 22-26 2002.
8. BATCHU, R., J. NEELAMEGAM, Z. CUI, M. BEDDHUA, A. SKJELLUM, Y. DANDASS, M. APTE. **Mpi/ftTM**: **Architecture and taxonomies for fault-tolerant, message-passing middleware for performance-portable parallel computing**. In Proceedings of the 1st IEEE International Symposium of Cluster Computing and the Grid held in Melbourne, Australia, 2001.
9. LOUCA, S., N. NEOPHYTOU, A. LACHANAS, P. EVRIPIDOU. **Mpi-ft: Portable fault tolerance scheme for mpi**. In Parallel Processing Letters, Vol. 10, No. 4, 371-382, World Scientific Publishing Company, 2000.
10. CHEN, Z., G. E. FAGG. **Building Fault Survivable MPI Programs with FTMPI Using Diskless Checkpointing**. Computer Science Department, University of Tennessee, USA, 2005.
11. PLANK, J. S., K. LI, **Faster Checkpointing with n+1 Parity**. In FTCS, 1994, pp. 288-297.
12. PLANK, J. S., **A Tutorial on Reed-Solomon Coding for Fault-Tolerance in RAID-like Systems**. Department of Computer Science University of Tennessee, Technical Report CS-96-332.2003.
13. CHEN, G. L., **Parallel Computing – Data Structure and Algorithm**, 1999.
14. DUFF, I., R. GRIMES, J. LEWIS **User's Guide for the Harwell-Boeing Sparse Matrix Collection**, October 1992.
15. DUFF, I., R. GRIMES, J. LEWIS, **Sparse Matrix Test Problems**. ACM Transactions on Mathematical Software, Volume 15, March 1989, pp. 1-14.
16. ANDREI, N., **Accelerated Conjugate Gradient Algorithm with Modified Secant Condition for Unconstrained Optimization**. Studies in Informatics and Control, vol. 18 (3), pp. 211-232, 2009
17. ANDREI, N., **A Hybrid Conjugate Gradient Algorithm for Unconstrained Optimization as a Convex Combination of Hestenes-Stiefel and Dai-Yuan.** Studies in Informatics and Control, ISSN 1220-1766, vol. 17 (1) , 2008, pp. 57-70.
18. ANDREI, N., **A Hybrid Conjugate Gradient Algorithm with Modified Secant Condition for Unconstrained Optimization as a Convex Combination of Hestenes-Stiefel and Dai-Yuan Algorithms.** Studies in Informatics and Control, ISSN 1220-1766, vol. 17 (4), 2008, pp. 373-392.
19. JAIRAM, N. K., K. V. KUMAR, N. SATYANARAYANA. **Scheduling Tasks on Most Suitable Fault tolerant Resource for Execution in Computational Grid.** International Journal of Grid and Distributed Computing, Vol. 5, No. 3, 2012, pp.121-132.
20. OSMAN, A., A. ANJUM, N. BATOOL, R. MCCLATCHEY. **A Fault Tolerant, Dynamic and Low Latency BDII Architecture for Grids.** International Journal of Grid and Distributed Computing, Vol. 3, No. 4, pp.1-18, 2010.