Proposal of MPI operation level checkpoint/rollback and one implementation

Yuan Tang
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee, Knoxville, USA
superTangcc@yahoo.com

Graham E. Fagg
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee, Knoxville, USA
fagg@cs.utk.edu

Jack J. Dongarra
Innovative Computing Laboratory
Department of Computer Science
University of Tennessee, Knoxville, USA
dongarra@cs.utk.edu

Abstract

With the increasing number of processors in modern HPC(High Performance Computing) systems, there are two emergent problems to solve. One is scalability, the other is fault tolerance. In our previous work, we extended the MPI specification on handling fault tolerance by specifying a systematic framework for the recovery methods, communicator, message modes etc. that define the behavior of MPI in case an error occurs. These extensions not only specify how the implementation of the MPI library and RTE (Run Time Environment) handle failures at the system level, but provide the normal HPC application developers with various recovery choices with varying performance and cost. In this paper, we continue the work on extending the MPI's capability in this direction. Firstly, we are proposing an MPI operation level checkpoint/rollback library to recover the user's data. More importantly, we argue that the future generation programming model of a fault tolerant MPI application should be recover-and-continue *against the more traditional* stop-and-restart *model.* Recover-and-continue *means that in case an error occurs, we just re-spawn the failed processes. All the remaining living processes stay in their original processors mapping on memory. The main benefits of* recover-and-continue *are much less cost for system recovery and the opportunity of employing in-memory checkpoint/rollback techniques. Compared with stable or local disk techniques, which are the only choices for* stop-and-restart*, doubtlessly, the in-memory approach significantly reduces the performance penalty in checkpoint/rollback. Additionally, it makes it possible to establish a concurrent multiple level checkpoint/ rollback framework. With the progress of our work, a picture of the hierarchy of future generation fault tolerant HPC system will be gradually unveiled.*

1 Background

The main goal of HPC is pursuing high performance. Confined by the performance that could be possibily achieved on single processor, HPC systems have progressed from single to multiple processor, and this trend will continue [10] [11]. Today's number 1 in the top500 list, the IBM BlueGene/L, is composed of 65536 processors. And $100K$ processor systems are in development [2]. With this trend of increasing the number of processors in one system , there are two emergent problems. One is scalability, i.e. whether the performance of HPC system could increase at the same pace as the increasing number processors. The other is fault tolerance. Concluding from the current experiences on top-end machines, a $100,000$ processor machine will experience multiple process failures every hour.

The current MPI Specification 1.2, the most popular parallel programming model, especially for large scale HPC systems, has not specified an efficient and standard way to process failures. Currently, MPI gives the user the choice between two possibilities of how to handle failures. The first one, which is also the default mode of MPI, is to immediately abort the application. The second possibility is just return the control back to the user application without requiring that subsequent operations succeed, nor that they fail. In short, according to the current MPI specification, an MPI program is not supposed to continue in case of error. While most systems currently are much more robust, even though partial node failure/unavailability are much more frequent, in most cases they will be recovered and brought back to the whole system quickly. So, there is a mismatch between hardware and the (non fault tolerant) programming model of MPI. There is a requirement for the programming model of MPI to include the handling of partial processes failure/unavailability.

In our previous work [5], we extended the MPI specification in this direction by specifying a systematic framework for the recovery procedures, communicator modes, message modes etc., i.e.

1. Define the behavior of MPI in case an error occurs. That is, FT-MPI, will recover the MPI objects and the execution context of the user application (NOT user data) in case a failure occurs.

2. Give the application the possibility to recover from a process failure. In addition to the standard ABORT, FT-MPI provides three more recovery/communicator modes.

    - *REBUILD*: re-spawn processes to the number before failures;
    - *BLANK*: just leave the failed/unavailable processes as holes in the system;
    - *SHRINK*: re-arrange the ranks of still living processes and pack them into a more compact rebuilt MPI_COMM_WORLD.

3. base it on MPI 1.2 (plus some MPI 2 features) with a fault tolerant model similar to what was done in PVM. That is, FT-MPI, working with the underlying HARNESS system [3], provides the failure detection and failure notification. Based on the user's choices, FT-MPI decides what are the necessary steps and options to start the recovery procedure and therefore change the state of the processes back to *no failure*

These extensions not only specify how the implementations of MPI library handles failures at system level, but provide the normal MPI application developers various recovery choices in between performance and cost. Also, an implementation of this extension, which is named FT-MPI [4], is available at `http://icl.cs.utk.edu/ftmpi`.

The main difference between FT-MPI's approach and a lot of other fault tolerant parallel systems is that FT-MPI adopts a programming model of *recover-and-continue* other than *stop-and-restart*, which is the tradition in lots of other fault tolerant parallel systems [1] [15] [7] [9] [16].

The main points of *recover-and-continue* are, when some processes are found failed/unavailable, the other still alive processes neither exit nor migrate. Instead, they stay in their original processor/memory mappings and will try re-spawning failed processes and re-building the communicator. From the system point of view, this approach significantly reduces the cost of RTE recovery (see [6] and Table 1). Also, it provides the opportunity to employ in-memory checkpoint/rollback techniques. More importantly, we could establish a framework of concurrent multiple level checkpoint/rollback on *recover-and-continue*.

But this previous work did not cover the users' data. In order to fully utilize the fault tolerant features of FT-MPI, user should write their own checkpoints and be responsible for rolling them back after the failure recovery of RTE (Run Time Environment).

2 Introduction

In this paper, we are continuing our efforts in extending the fault tolerant capability of MPI.

We are proposing an MPI operation level checkpoint/rollback standard, which, in our opinion, is a trade-off approach in between traditional system level automatic checkpointing and user level manual checkpointing.

The main rationals for MPI operation level checkpoint/rollback are:

- *Portability:* the user fault tolerant application written by this interface could be guaranteed to run correctly across different platforms without any changes to their source code. This is the main drawback of traditional user level manual checkpointing because not all the checkpoint/rollback techniques are available on all platforms.

- *Software re-use:* the implementor of the MPI library could integrate various checkpoint/rollback techniques in the library for all the applications to share. This is also the main drawback of traditional user level manual checkpointing, which requres every application to re-implement the checkpoint/rollback techniques repeatedly.

- *High performance:* the user specifies which data to checkpoint and which data could be computed from the data to checkpoint. The total size of the checkpoint is significantly reduced, which will doubtlessly outperform any system level automatic approach.

- *Software hierarchy:* All the future changes in processing checkpoint/rollback will be limited to the MPI library and the end user could focus solely on their special problem area.

Also, we argue that the programming model of the fault tolerant MPI application should be *recover-and-continue*.

In *recover-and-continue*, when some failure occurs, only the failed processes will be re-spawned and might be migrated to other processors. Other still alive processes neither exit nor migrate. Instead, they stay in their original processor/memory mappings . From the point view of checkpointing, this model provides the opportunity of employing in-memory (diskless [14]) checkpoint/rollback techniques, more importantly, the opportunity for establishing a framework of concurrent multiple level [17] checkpoint/rollback.

In summary:

1. The specification proposal is based on MPI. The implementation is currently based on FT-MPI [5];
2. It provides a standardized method and uniformed interface for end users to write their fault tolerant MPI applications.
3. The implementation is Two-level [17]. It employs the *imem-m-rep* algorithm (subsection 5.2) preparing for at most $m, where m \leq (n-1)$ number of simultaneous failures. We use a much longer periodic *stable-disk* algorithm to prepare for multiple copies of checkpoint as well as the rare, but fatal n total process failures. The implementation allows the user to specify the ratio of the percentage of checkpoints taken in-memory and the percentage stored on stable disk, while the actual switch between these two levels are dynamic and transparent to the user.
4. It supports all MPI data types.
5. Since FT-MPI is responsible for recovering the RTE, MPI objects, and internal message queues, this MPI operation level checkpoint/ rollback library covers only the users application data.
6. With the current implementation based on FT-MPI, we have performed some performance tests, which provide a good and quantified reference for writing fault tolerant applications.

3 MPI operation level checkpoint and rollback

Before starting the detailed description of our efforts, we will define and clarify the exact meaning of some frequently used terms in the rest of this paper.

- n: is the total number of processes in system, including both dead and alive processes.
- m: is the number of failures we are going to tolerate simultaneously, ie. within the period of one round of recovery. Additionally, we assume the condition $m \leq n - 1$ always holds.
- nof: number of failures, equals m.
- *imem-m-rep*: The algorithm of "in memory m replication", which will be discussed in subsection 5.2.
- *stable-disk*: The checkpoint/rollback algorithm of writing to and reading back from a stable disk system.
- *old process*: The process which has experienced at least one round of recovery and is still currently alive.
- *new process*: The newly re-spawned process.
- *RTE*: Run Time Environment.

3.1 Specification Proposal

In order to provide a standard method and uniformed interface for an MPI application developer to write checkpoints and roll back , the FT library should provide:

1. *MPI_Ckpt_open(MPI_Ckpt_options * options)*: Initialize the necessary data structure and do some preparation work for checkpoint and rollback. And this function should be called after MPI_Init().
   - The idea here is to make checkpoint and rollback as easy and standard as reading or writing a normal UNIX file. Also, the separation of MPI_Ckpt_
     open() from MPI_init() is to give the user a choice of not to introduce the checkpoint and rollback interface and cost into their applications.
   - The data structure MPI_Ckpt_options allows the library implementor some freedom for providing implementation specific options and choices. For example, which algorithm or level they prefer, how many failures he plans for the system to tolerate simultaneously, the ratio for two (2)-level switching, etc. In order to maintain consistency, the options specified in MPI_Ckpt_options should have global effect. That is, if the user specifies one particular checkpoint algorithm, this algorithm will work on all the data to be checkpointed; if the user specifies the ratio of 2 levels, the ratio will remain a global constant until it is explicitly changed by another function call, etc.
2. *MPI_Ckpt_close()*: Counterpart of MPI_Ckpt_open().
3. *MPI_Ckpt(void * data_needs_ckpt, long length, MPI_ Datatype datatype, int tag)*: This is a registration function. The user could use it to mark which data to checkpoint and separating it from the data could be easily computed from the data to checkpoint. Thus the size of the checkpoint could be controlled. Also,

the checkpoint and rollback library should support any MPI data types.

4. *MPI_Ckpt_here()*: This function is the one which will do the actual work. Users call this function at some synchronized points in their program to checkpoint all the data previously registered by MPI_Ckpt(...). The idea is to let the user make the decision where to checkpoint :

   - We have the general rule: the user knows his application the best. This is also the foundation of lots of user manual fault tolerant algorithms.
   - The operation or algorithm (e.g. like Chandy / Lamport) to get a global synchronous status is very expensive.
   - Due to the fact that an MPI application might do no synchronous work after its call to MPI_Init(), the global synchronous status from the system point of view might still be different from the user application's point of view.

5. *MPI_Rollback(void * data_rollback, long length, MPI_Datatype datatype, int tag)*: Users call this function after RTE recovery to rollback the data from the latest complete checkpoint copy. Also, he has the freedom to rollback those data in random order.

6. *MPI_Remove_ckpt(int tag)*: If some data are no longer in use , the user could call this function to prevent them from future checkpointing and save the performance.

7. *MPI_Ckpt_ctrl(MPI_Ckpt_options * options)*: If necessary, user could call this function to change the global checkpoint and rollback options. The idea here is to make the checkpoint and rollback component as easy to control as the UNIX file or I/O system.

4 Sample Pattern

In this section, we are providing a sample pattern (see Figure. 1) to show how to write a fault tolerant application on top of FT-MPI with the hope it can provide a standard and uniformed way of integrating checkpoint and rollback features into normal MPI applications.

5 Current Implementation of FT-MPI checkpoint and rollback library

Based on the above specification proposal and current implementation of FT-MPI, we implemented the MPI operation level checkpoint/rollback library.

```
void  error_handler(MPI_Comm * pcomm, int * prc, ...){
    recover_comm(pcomm);
    MPI_Error_string(*prc, errstr, &len);
    longjmp(here, 1); /* escape from hell */
}
void  main(){
    struct data_type data_needs_ckpt[];
    struct data_type data_could_comp[];

    MPI_Init();
    MPI_Ckpt_open(MPI_Ckpt_options)
    MPI_Errhandler_create();
    setjmp(here); /* after recovery, longjmp here */
    /* Set the error handler to the comm world */
    MPI_Errhandler_set();

    if (after_recovery)
       MPI_Rollback(data_needs_ckpt,length,datatype,tag);
    else/* normal startup */
       MPI_Ckpt(data_needs_ckpt,length,datatype,tag);
    for(;;){
    Compute(data_needs_ckpt);
    Compute(data_could_comp,data_needs_ckpt);
    MPI_Ckpt_here(); /* Sync point */
    /* if necessary, change the ctrl options */
    MPI_Ckpt_ctrl(MPI_Ckpt_options);
    }
    MPI_Ckpt_close();
    MPI_Finalize();
    return;
}
```

Figure 1. Sample Pattern of integrating the feature of checkpoint and rollback into MPI programs

5.1 Possible optimizations

From [14] [12] [13] [8] and other checkpoint and rollback related papers, we analyzed the main performance bottleneck of checkpointing and provide some possible optimization methods.

1. Size of checkpoint. With MPI operation level checkpointing, users could specify which data to checkpoint. The size of checkpoint might be significantly reduced.
2. The internal collective communication introduced by checkpointing. In FT-MPI, we use dynamic switching techniques to select from several known and implemented collective algorithms according to the message size, communicator mode, number of process involved in communication, etc. So the collective, especially the *Reduce*, *Bcast*, *Allgather* algorithms used in getting the checkpoint, global status, and rollback have been optimized.
3. The storage media user to store the checkpoints.
   - According to the availability of local memory, remote nodes, and external stable disk, the priority of the data to checkpoint, the number of failures to tolerate simultaneouly, the checkpoint routine dynamically switches from storage media to media to find a tradeoff in between performance and robustness (see Figure 4).
   - Utilize the local memory or the memory of remote nodes to prepare for the more frequent and less fatal errors, and use a longer periodic write-to-stable-disk strategy preparing for the worst case, i.e. all processes down or an error occurs in the middle of checkpointing. That is, an implementation of the *Two-Level Recovery Scheme* [17]. So a deliberate selection of the ratio of 2 level becomes very important in getting the tradeoff between costs, performance and robustness.
4. By default, there are one in-memory copy and two on-stable-disk copies of checkpoint (all these 3 copies of checkpoint are of different time step) co-existing in the system for robustness.

5.2 The imem-m-rep algorithm

We implement an *in-memory-m-replication* algorithm to store one copy of checkpoint in local redundant memory on each node. This algorithm is designed to tolerate any $m, where m \leq n-1$ number of process failure/unavailable simultaneously. Figure 2 demonstrates when the *nof* equals 2, i.e. $m == 2$, how the checkpoint process of *imem-m-rep* works.

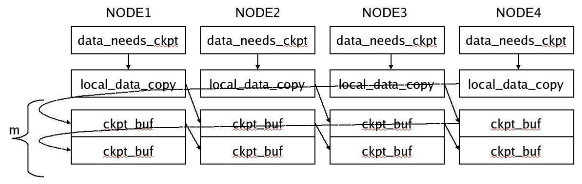

Figure 2. How imem-m-rep-ckpt works – Assume m equals 2

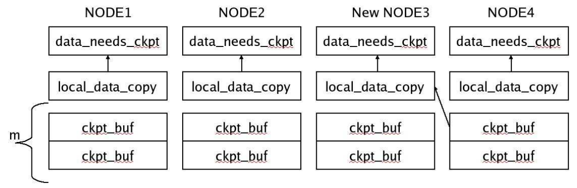

Figure 3. How imem-m-rep-rollback works – Assume the Process on Node 3 is newly re-spawned

1. When the MPI_Ckpt_here() routine starts, every process (assume its rank is i) first makes a copy of the *data_needs_ckpt* into *local_data_copy*
2. Every process i signals the background checkpoint thread to run. Then the main thread will return the control flow back to user
3. The background checkpoint thread of process i sends the data from *local_data_copy* to the *ckpt_buf* of process $(i+1)\%n$
4. If $m > 1$, every process i will continuously send the data received from its *PREVIOUS* (process $(i-1+n)\%n$) to its *NEXT* (process $(i+1)\%n$).
5. The send/receive pipeline continues until the number of loops equals m.
6. When the number of loops equals m, the checkpoint thread stop working and write one bit in corresponding data structure to signal the main thread that current round of checkpoint is completed.

With this checkpoint algorithm, the rollback algorithm is straight forward. When any up to m of the processes failed and re-spawned,

1. All the processes step into a stage of global status gather by MPI_Allgather(). Every process, old or new, will then know how many processes died in the last round of failure and who is new, as well as whether

the last round of *imem-m-rep* checkpointing has completed in all the processes.

2. If the last round *imem-m-rep* checkpoint process has completed in all the old processes and the number of process failed is less than or equal to the previously set *nof*,

   every newly re-spawned process i will calculate who is its *NEXT* old neighbour and could get its checkpoint back. Then it post a receive request to it.

   all the old processes will employ the same calculation algorithm to know who should be responsible for its nearest *PREVIOUS* new process. Then the responsible one will post a send to the new ones.

3. Else

   it will call the *stable-disk* rollback algorithm

4. Only every new MPI process' nearest *NEXT* will send one copy of corresponding checkpoint data in its *ckpt_buf* to the new ones.

5. All the other processes just rollback from its *local_data_copy* without any communication.

The rollback procedure is illustrated in Figure 3.

Obviously, this *imem-m-rep* algorithm could tolerate any $m; m \leq n-1$ number of process failures simultaneously.

The main advantage of this algorithm is the low cost during rollback: only every newly re-spawned process and its nearest *NEXT* are involved in communication. All the other old processes could rollback by a simple memcpy() , which is very fast. The quantified rollback differences of these three types of process could be observed in Table 2.

The main disadvantage of this algorithm is the memory consumption problem. Memory is very precious in large scale scientific computing. So we employ one more algorithm of writing to and reading back from stable disk.

5.3 The stable disk algorithm

The stable disk algorithm

1. Make multiple copies of checkpoint. To save memory, the *imem-m-rep* algorithm only has one copy of checkpoint. So in case an error occurs in the middle of processing the checkpoint, we should utilize the stable disk.

2. The stable disk algorithm has two copies of checkpoint (each copy is of different time step) for robustness. When the rollback procedure of the stable disk is invoked, it will rollback from the latest coherent copy.

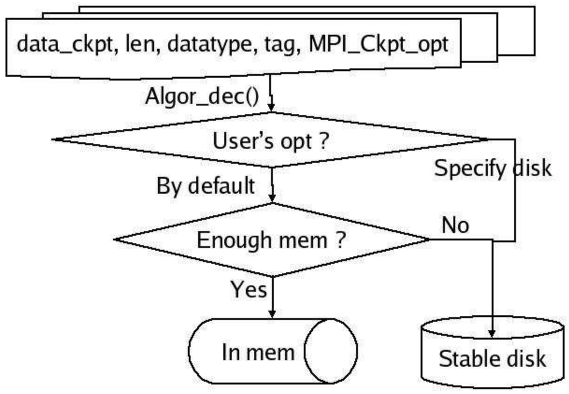

Figure 4. How MPI_Ckpt() works

3. It makes our checkpoint/rollback system two (2) level. The *imem-m-rep* algorithm is prepared for more frequent and less fatal $m; m \leq n-1$ number of process failure, while the stable-disk algorithm is always ready for the worst.

4. Due to the much higher overhead of the stable disk algorithm (which could be observed in Table 1 and Table 2 of section 6) by default, in our current *MPI_Ckpt_here()* version, the ratio of stable disk algorithm to imem-m-rep algorithm is set to $1 : 1000$. That is, we invoke the stable disk checkpoint routine every 1000 times of imem-m-rep. Of course, the user has been granted the right to change this ratio by changing the input parameter *MPI_Ckpt_options* of *MPI_Ckpt_open()* or *MPI_Ckpt_ctrl()* functions.

5.4 How MPI_Ckpt() works

Figure 4 illustrates how our MPI_Ckpt() dynamically switches between different levels of checkpoint algorithms and implmentations depending on the user's input parameters and the available resources in system. For example, is there enough memory to tolerate *nof* number of simultaneous failures in *imem-m-rep* algorithm, etc.

6 A Preview of Testing Data

Due to page limit restrictions, we only attach some typical and interesting test results of the FT-MPI checkpoint/rollback library here.

6.1 Testing Platform – TORC

Our testing platform (TORC) is a collaborative effort between the University of Tennessee's Innovative Computer Laboratory in the Computer Science Department and Oak

Ridge National Laboratory. It is comprised primarily of commodity hardware and software.

- Hardware:
  - Myrinet 8-port switches, and PCI LANI 4.1 cards
  - 2×16-port Fast Ethernet Switch (Bay Networks 350T)
  - Compute nodes:
    * Dual 933MHz Pentium III (256KB cache)
    * Dell WS400 machines, using the PCI 82440FX (Natoma) chipset
    * 512 MB RAM
    * 3Com Fast Etherlink 905TX 10/100 BaseT Network Interface Card (integrated)
- Software:
  - Red Hat Linux (2.4.22 multiprocessing kernel)
  - Gnu C/C++ (g++ 3.3.2)

6.2 Sample Testing Data

Here, we provide some explanation of Table 1 and Table 2:

- The *Sender*, *Recver* and *Normal* row in Table 2 stands for the overhead of the old process who would send one copy of its checkpoint data to its *PREVIOUS* new process, the new process who would receive one copy of checkpoint from its nearest *NEXT* neighbour, and all the other old processes who would rollback from local "ckpt_buf" without any communication, respectively. The details of this rollback procedure could be found in subsection 5.3. The rollback overhead of these three types of processes varies, so they are listed separately.
- Due to the set of full buffer mode on the stable disk file, the rollback overhead of old processes and newly re-spawned processes differ, so only the rollback overhead of the new processes are listed.
- In *imem-m-rep* algorithm, the data shown is for $nof = 1$.
- All the testing results in Table 1 and Table 2 are those for four processes on four different nodes.
- The $400B, 40KB, and 4MB$ in the most left column of both Tables mean the corresponding row of testing results are received from checkpoint size 400Bytes, 40KBytes and 4MBytes data, respectively.

Comparison between the Table 1 and Table 2, we could conclude:

	T_recover loop	T_stable ckpt	T_ckpt total	T_stable rollback	T_rollback total
400B OLD	0.6684	0.0151	0.0185	0.0033	0.0051
400B NEW	-	-	-	0.0040	0.0094
40KB OLD	0.6434	0.0295	0.0356	0.0033	0.0047
40KB NEW	-	-	-	0.04767	0.06037
4MB OLD	-	3.0704	3.2319	0.0380	0.0453
4MB NEW	-	-	-	1.5401	1.5781

Table 1. Checkpoint and rollback overhead of stable disk algorithm

	T_imem ckpt	T_ckpt total	T_imem rollback	T_rollback total
400B Sender	0.0026	0.0062	0.000151	0.0016
400B Recver	-	-	0.000151	0.0015
400B Normal	-	-	0.000004	0.0046
40KB Sender	0.0261	0.0269	0.000893	0.002584
40KB Recver	-	-	0.001431	0.037701
40KB Normal	-	-	0.000143	0.035958
4MB Sender	0.6907	1.0701	0.353990	0.391026
4MB Recver	-	-	0.344040	0.381836
4MB Normal	-	-	0.022379	0.059456

Table 2. Checkpoint and rollback overhead of imem-m-rep algorithm

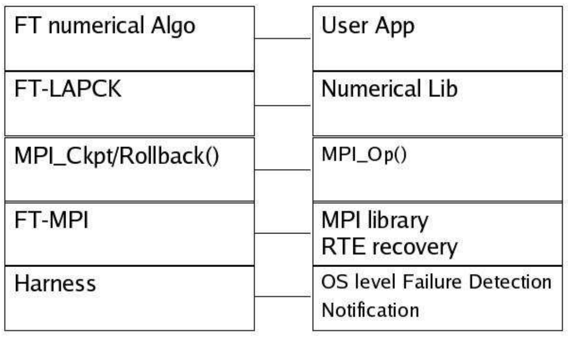

Figure 5. Hierarchy of future generation fault tolerant parallel system

1. Both overheads of the *stable-disk* algorithm and *imem-m-rep* increases significantly as the size of checkpoint increases.

2. When the checkpoint size is very small, such as the 400Byte row, the overhead of the *stable disk* algorithm is more than ten times higher.

3. The performance of the *stable disk* checkpoint and rollback algorithm is restricted by the bus or efficiency of Parallel I/O. The power of the *imem-m-rep* algorithm is limited by memory and network.

7 Hierarchy

With the progress of our work, a picture of the hierarchy of future generation, fault tolerant parallel systems becomes more and more clear as Figure 5 illustrates.

From the bottom level HARNESS [3], which is responsible for the failure detection and notification; upper level FT-MPI, which takes a systematic procedure/steps to recover the MPI objects, run time environment and user application context; For upper level applications, FT-MPI provides the MPI_Ckpt/Rollback() routines. Eventually, the most upper level user application and fault tolerant numerical library (e.g.FT-LAPACK) will be built upon all these underlying facilities and benefit from them.

References

[1] G. Burns, R. Daoud, and J. Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[2] J. Dongarra. An overview of high performance computers, clusters, and grid computing. *2nd Teraflop Workbench Workshop*, March 2005.

[3] G. E. Fagg, A. Bukovsky, and J. J. Dongarra. Harness and fault tolerant mpi. *Parallel Computing*, 27(11):1479–1495, 2001.

[4] G. E. Fagg and J. Dongarra. Ft-mpi: Fault tolerant mpi, supporting dynamic applications in a dynamic world. In *Proceedings of the 7th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 346–353, London, UK, 2000. Springer-Verlag.

[5] G. E. Fagg, E. Gabriel, G. Bosilca, and et al. Extending the mpi specification for process fault tolerance on high performance computing systems. *Proceedings of the ISC2004*, June 2004.

[6] G. E. Fagg, E. Gabriel, Z. Chen, T. Angskun, G. Bosilca, J. Pjesiva-Grbovic, and J. J. Dongarra. Process fault tolerance: semantics, design and applications for high performance computing. *The International Journal of High Performance Computing Applications*, 19(4):465–477, 2005.

[7] E. Godard, S. Setia, and E. L. White. Dyrect: Software support for adaptive parallelism on nows. In *IPDPS '00: Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, pages 1168–1175, London, UK, 2000. Springer-Verlag.

[8] K. Li, J. F. Naughton, and J. S. Plank. Low-latency, concurrent checkpointing for parallel programs. *IEEE Trans. Parallel Distrib. Syst.*, 5(8):874–879, 1994.

[9] V. K. Naik, S. P. Midkiff, and J. E. Moreira. A checkpointing strategy for scalable recovery on distributed parallel systems. In *Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–19, New York, NY, USA, 1997. ACM Press.

[10] T. Organization. System processor counts/systems in top500 list nov. 2004. *http://www.top500.org/lists/2004/11/charts.php?c=12*, November 2004.

[11] T. Organization. System processor counts/systems in top500 list june 2005. *http://www.top500.org/lists/2005/06/charts.php?c=12*, June 2005.

[12] J. S. Plank. A tutorial on reed-solomon coding for fault-tolerance in raid-like systems. *Softw. Pract. Exper.*, 27(9):995–1012, 1997.

[13] J. S. Plank and Y. Ding. Note: Correction to the 1997 tutorial on reed-solomon coding. *Softw., Pract. Exper.*, 35(2):189–194, 2005.

[14] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Transactions on Parallel and Distributed Systems*, 9(10):972–??, 1998.

[15] J. M. Squyres and A. Lumsdaine. A Component Architecture for LAM/MPI. In *Proceedings, 10th European PVM/MPI Users' Group Meeting*, number 2840 in Lecture Notes in Computer Science, pages 379–387, Venice, Italy, September / October 2003. Springer-Verlag.

[16] S. S. Vadhiyar and J. Dongarra. Srs: A framework for developing malleable and migratable parallel applications for distributed systems. *Parallel Processing Letters*, 13(2):291–312, 2003.

[17] N. H. Vaidya. A case for two-level recovery schemes. *IEEE Trans. Comput.*, 47(6):656–666, 1998.