Algorithm-Based Master-Worker Model of Fault Tolerance in Time-Evolving Applications

Md. Mohsin Ali and Peter E. Strazdins
Research School of Computer Science
The Australian National University
Canberra, ACT 0200, Australia
{md.ali, peter.strazdins}@anu.edu.au

***Abstract*—In order to make the High Performance Computing (HPC) applications fault-tolerant, many application developers are investigating Algorithm-Based Fault Tolerance (ABFT) techniques to improve the efficiency of these applications recovery beyond what existing checkpoint/restart techniques alone can provide. Unfortunately, the standard library Message Passing Interface (MPI) used for implementing this type of application do not have standardized fault tolerance semantics. This paper presents how the fault tolerance semantics of Fault-Tolerant MPI (FT-MPI) can be used as a part of ABFT to design and implement a fault-tolerant algorithm applicable for time-evolving applications which could survive process failures. The model of the presented technique is a master-worker scheme which can tolerate the failures of all worker processes. As an example of time-evolving application, we consider the upwind scheme of one dimensional advection equation solution. We focus on communication-level issues, data prevention techniques, as well as time-evolving control issues. This paper also highlights a common set of issues including failure detection, failed process recovery, duplicate message handling, etc. This contribution will help application developers to resolve different issues of design and implementation of fault-tolerant algorithms for more complex time-evolving applications.**

Keywords-fault tolerance; MPI; FT-MPI; process failure;

I. INTRODUCTION

Today's High Performance Computing (HPC) systems use hundreds of thousands of processing elements to concurrently execute millions of threads and this number is increasing day-by-day. The computational clusters composed of such multiple processing elements called cores are connected with high-speed networks designed to minimize the communication costs and maximize reliability, see for example [1]. A concerted effort is required in order to exploit the full performance of these new computational clusters. This performance is critically needed in areas like climate and environmental research and in physics and energy research characterized by complex scientific models. The most common such models use the solution of systems of partial differential equations in an iterative way. The time-evolving solution of simple one dimensional advection equation is a very basic one among them and is discussed in [2].

Besides exploiting the full performance of such large clusters, a critical issue is how to deal with hardware and software faults that lead to process failures. The failure rate of a system is roughly proportional to the number of processor elements in that system [3]. For instance, a recent study shows that in a particular model of the Blue Gene system located at the Oak Ridge National Laboratory, a 100,000-processor machine experiences a processor failure every few minutes [4]. Since the size of the HPC systems are becoming larger, as we mentioned before, the failure rates of these large systems are increasing day-by-day [5].

The Message Passing Interface (MPI) [6] specification, which is widely used as a parallel programming paradigm for HPC, could not deal with one or more process failures at run-time. Generally, MPI provides two options for handling failures.

- The first option with error handler `MPI_ERRORS_ARE_FATAL`, which is also the default mode of MPI, is to immediately abort the application.
- The second option, which uses error handler `MPI_ERRORS_RETURN`, is just slightly more flexible; handing the control back to the user application without guaranteeing that any further communication can occur. Its purpose is to mainly give an application developer the option to perform some local operations before exiting.

Another important challenge in HPC for dealing with the issues of fault tolerance is the deficiency of availability of both theoretical and practical literature to get an idea about the range of issues during the development of the fault-tolerant program. Besides this, there is a discrepancy between the capabilities of current HPC systems and the most widely used parallel programming paradigm (MPI). Although the MPI specification proves itself for fully exploiting the capabilities of the current architectures, it can not handle the failure of processes similarly. As a result, one of the main reasons why many researchers prefer Fault-Tolerant MPI (FT-MPI) [7] as an interface to implement their applications is because of its capability to handle process

Copyright (c) IARIA, 2013.

failures in run-time. It is actually an MPI-1 implementation that extended the MPI communicator states and modified the MPI communicator construction functions. Details of this are discussed in Section III.

The contributions of this paper are as follows:

- Design and implement a fault-tolerant algorithm applicable for time-evolving applications which could survive process failures.
- The presented model is a master-worker model which could survive the failure of all workers in the system.
- The failed processes are rebuilt including the recovery of their data.
- Presenting how the fault tolerance semantics of FT-MPI can be used for failure recovery as a part of an ABFT technique.
- This is a very basic model which is currently not scalable, but there is scope of modifying this model to make it scalable.

The rest of the paper is organized as follows. Related research work is discussed in Section II. Section III describes the semantics and interfaces of FT-MPI that are used for implementing fault-tolerant MPI applications surviving process failures. Section IV describes a fault-tolerant version of time-evolving solution of one-dimensional advection equation demonstrating the techniques of detection and recovery of process failure, recovery of lost data, handling of duplicate messages, and controlling of iteration after the failure. Performance comparison of Open MPI with FT-MPI and experimental results demonstrating failure recovery performance of FT-MPI is provided in Section V. Finally, concluding remarks for this paper are given in Section VI.

II. Related Work

A master-worker model of MPI programs that could recover from process failure by using multiple intercommunicators is proposed in [8]. In this model, the management of multiple sets of intercommunicators for a single group of processes is cumbersome in comparison to directly using a single set of intracommunicators. Moreover, this model is not used for time-evolving applications.

A fault-tolerant time-evolving program applicable for ring type communication is proposed in [9]. The focus of this work is on the run-through stabilization component of the developing proposal which is being extended to include flexible recovery strategies of MPI Forum's Fault Tolerance Working Group [10]. The run-through stabilization component of the proposal provides an application with the ability to continue running and using MPI even when one or more processes in the MPI universe fail, but failed processes become permanently unresponsive to communications. Moreover, data recovery issues are not considered in this component. So, this approach will not be applicable for the systems which require the recovery of the lost data due to process failure.

An algorithm-based fault tolerance technique using checksum for detecting and recovering one error in HPC is proposed in [11]. This is applicable for specific problems like parallel matrix-matrix multiplication. Other fault-tolerant algorithms related to matrix operations are available in [12] and [13].

A floating-point arithmetic coding approach into diskless checkpointing is proposed in [14] to address the associated round-off errors. This approach could survive only a small number of process failures and could not survive all process failures.

A natural fault-tolerant algorithm for iterative problems is proposed in [15] where the algorithm computes a new approximate solution from the data of the non-failed processes after the failure. The main drawback of this approach is that the convergence after failure of the processes is no longer the same as the original method. Moreover, this algorithm is also not applicable for the case where it needs actual solution.

An algorithm-based recovery approach for iterative methods is proposed in [16] where neither a checkpoint nor a roll-back is necessary for recovering the data of the failed processes. It demonstrates that, for many iterative methods, if the parallel data partition scheme satisfies certain conditions, the iterative methods themselves can maintain enough inherent redundant information to tolerate failures in the computation. Under this condition, the computation can be restarted from where the failure occurs without any checkpointing. Although this approach is scalable, it cannot be used for generalized iterative problems.

III. FT-MPI Semantics and Interfaces

Since current semantics of MPI could not guarantee further communication to occur after a failure, as we mentioned before, we need to modify its semantics so that it can take some corrective actions to rebuild the communicator with an aim to continue the communication after detecting a failure. FT-MPI is such an MPI-1 implementation, as we mentioned before, that extended the MPI communicator states and modified the MPI communicator construction functions. The modified semantics of FT-MPI include state of the communicator, state of the process, mode of communicator, mode of the message, etc. As for example, FT-MPI extends the MPI communicator states from {`valid, invalid`} to a range {`FT_OK, FT_DETECTED, FT_RECOVER, FT_RECOVERED, FT_FAILED`}. It can be usually described as {`OK, PROBLEM, FAILED`}, whereas the remaining are used for the internal fault recovery algorithm of FT-MPI. Similarly, typical states of MPI processes called {`OK, FAILED`} are replaced by {`OK, Unavailable, Joining, Failed`} in FT-MPI.

A communicator in FT-MPI changes its state after detecting a probable error when either

- an MPI process changes its state due to its failure or anything else, or
- a communication within that communicator fails for some reason.

For the first case, all communicators that include this process are changed. Whereas for the second case, not all communicators are forced to be updated. Changing the communicator state includes rebuilding that communicator in order to recover from the probable error. A modified version of one of the communicator functions e.g. `MPI_Comm_{create, split or dup}` is used for this rebuild. Depending on the mode of failure, a newly built communicator can hold several modes such as `SHRINK`, `BLANK`, `REBUILD`, and `ABORT`. In order to keep the data structure contiguous, the communicator is reduced by `SHRINK` mode to fill the rank(s) of failed process(s) by the rank of the following process(es). So, the ranks of the processes are changed and forcing the application to recall `MPI_COMM_RANK`. `BLANK` is similar to `SHRINK` except that the communicator can contain gaps where there were problems in the processes during communication. These gaps can be filled later when necessary, but communicating with a gap causes an invalid rank error. Moreover, `MPI_COMM_RANK` returns the total number of processes including failed one which is no more valid. The more complex mode is `REBUILD`, which forces the creation of new processes to fill all the gaps containing empty ranks of the communicator. The new processes can be placed either in the empty ranks, or the communicator can be shrunk at first and then the remaining processes filled at the end. The last mode `ABORT` forces the application to abort immediately after detecting an error and there is nothing to do for a user. Example 3 of [17] shows how a communicator is simply rebuilt and reused when the communicator detects an error.

The MPI standard does not return additional error codes and classes except standard ones. But FT-MPI notifies the process failure once the application attempts to communicate directly (e.g., point-to-point operations) or indirectly (e.g., collective operations) with the failed process through the return code called `MPI_ERROR_OTHER` of the function, and error handler set on the associated communicator. This return code also makes additional information available via the *attribute caching mechanism* including a human readable form [17]. The first form returns the error information for a complete communicator in terms of the number of failures per rank (example 2 of [17]) since last recovery. The second form returns the failed ranks in the same order as they happened locally (example 1 of [17]). Using these information, an application developer can write down a fault-tolerant program to handle the error from the user level. Other than this, communications within a communicator is controlled by a message mode and can be either `NOP` or `CONT`. For `NOP`, there is nothing to do from the user level and allows the application to return from any point in the code to a state where it can take appropriate action as soon as possible based on the error. On the other hand, with `CONT` mode, all communication consisting of unaffected processes can continue as normal and attempts to communicate with a failed process reports an error until the communication state is reset. A sample FT-MPI master-worker code is available in example 4 of [17], where the communicator mode is `BLANK` and the communication message mode is `CONT`. The master keeps track of the work allocated and on an error it checks whether there is any surviving workers remaining or not. If any of these are available, then it just reallocates the work to them to continue the computation.

IV. Time-Evolving Applications: One-Dimensional Advection Equation Solution

The solution of advection equation is an important subject in scientific HPC. There are two reasons behind this. Firstly, advection is a part of important applications of HPC: meteorology, climatology, and air pollution. Secondly, many of the computations done on HPC systems involve the solution of partial differential equations. Since advection equation is actually a relatively simple partial differential equation, it provides a good starting point to study a broad class of computations performed on supercomputers.

There are broad classes of advection equations ranging from simple to complex. The one-dimensional advection equation is the most basic one among them. Algorithms for solving such basic equations are available in [2], along with the definition and uses of ghost values. The principle of these algorithms are like that the original advection values are divided into parts and then distributed them into a number of processors, say n processors. Then in each iteration, the following activities are performed.

- Each processor updates their ghost values by exchanging messages with their *left* and *right* neighbors (*left* of processor 1 is processor n and *right* of processor n is processor 1), see Fig. 1.
- Each processor computes their flux values and update their advection values according to the type of advection equation.

Finally, at the end of the iteration, the computed advection values from each of the processors are combined to generate the actual advection values.

Let us discuss the fault-tolerant version of this algorithm. Literally, there are many meanings of "fault-tolerant". See [8] for details. But in this paper, by fault-tolerant, we mean tolerance of process failure and this failure may happen for any reason. We consider a process failure as a *fail-rebuild-working* failure, that is, failed processes will be rebuilt and become available to communications. Since the data of a failed process is lost, this algorithm also recovers this lost data.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-268-4

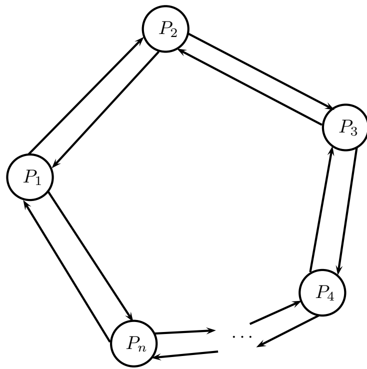

Figure 1. Communication in non-fault-tolerant advection equation solution.

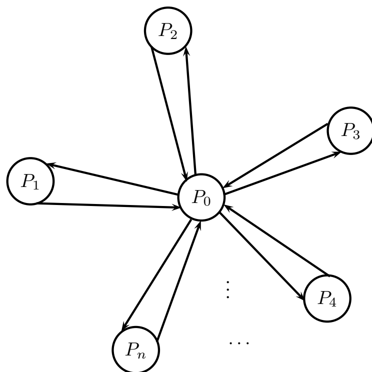

Figure 2. Communication in fault-tolerant advection equation solution.

The model of our fault-tolerant algorithm is master-worker where communications between a worker to its neighbors are done through master similar to Fig. 2 when process P_0 serves as master. In order to update the ghost values for each worker by exchanging messages with *left* and *right* neighbors (see Fig. 1), the following activities are performed.

- The master receives messages containing advection values from all workers and stores them in memory.
- The master calculates *left* and *right* ghost values for each worker from these stored values and send these calculated values to corresponding workers to update their ghost values.

Storing these values have additional benefits which are discussed in Section IV-C. However, it is easily observed that the total number of communication for updating ghost values of the workers through master is the same as that of without master except the increased size of the exchanged messages. See Fig. 1 and 2 for comparison.

The fault-tolerant algorithm for solving one-dimensional advection equation is shown in Fig. 3. The main points of this algorithm are as follows.

- A user-defined error handler is registered on Lines 8 and 9 (details in the algorithm shown in Fig. 8) for handling error including process failure and rebuilding the failed processes.
- Original advection values (input) generation and distributing them to workers are done on Line 11.
- Saving the values of workers to master is done on Line 13 such that the master can provide these values to workers when they rebuild after the failure.
- The main iteration is going on between Lines 16–22.
- The master is receiving advection values from all the workers on Line 18 (details in the algorithm shown in

```
Function int main(int argc, char *argv[])

1:  /* Initialize MPI */
2:  MPI_Comm MCW = MPI_COMM_WORLD;
3:  MPI_Errhandler errh;
4:  int rc_init = MPI_Init(&argc, &argv);
5:  MPI_Comm_rank(MCW, &process_id);
6:  MPI_Comm_size(MCW, &procs);
7:
8:  MPI_Errhandler_create(recover, &errh);
9:  MPI_Errhandler_set(MCW, errh);
10:
11: input_generate_and_distribute();
12:
13: save_values_in_master();
14:
15: /* Main Iteration */
16: for (i = 0; i < MAX_TIME_STEPS; i++) do
17:     if (process_id == MASTER) then
18:         master_rcv_activity(procs);
19:         ghost_iter_activity(procs);
20:         master_send_activity(procs);
21:     else // process_id == WORKER
22:         worker_activity(process_id);
23:
24: master_collects_distributed_values();
25:
26: MPI_Finalize();
27: return 0;
```

Figure 3. Main function of the algorithm.

```
Function void master_rcv_activity(int procs)

1:  for (j = 1; j < procs; j++) do
2:      do
            master_is_receiving_from_worker(j);
3:          if (rc_init == MPI_INIT_RESTARTED_NODE) then
4:              any_worker_re-spawned = 1;
        while (receiving is not SUCCESSFUL);
```

Figure 4. Master is receiving from workers.

```
Function void ghost_iter_activity(int procs)

1:  /* updating ghost values */
2:  for (j = 1; j < procs; j++) do
3:      if (any_worker_re-spawned == 1) then
4:          /* Load previous advection values */
5:          copy prev_advection_values(j) to advection_values(j);
6:      calculate_ghost_values_for_j(advection_values);
7:
8:      /* Saving advection values */
9:      save advection_values(j) to prev_advection_values(j);
10: if (any_worker_re-spawned == 1) then
11:     /* reset any_worker_re-spawned */
12:     any_worker_re-spawned = 0;
13:
14:     /* load previous time step */
15:     i - -;
```

Figure 5. Updating ghost values and saving/restoring advection values.

```
Function void master_send_activity(int procs)

1: for (j = 1; j < procs; j++) do
2:     do
3:         master_is_sending_to_worker(j);
       while (sending is not SUCCESSFUL);
```

Figure 6. Master is sending to workers.

Fig. 4). Received values in the master may come from processes which are just rebuilt after failure. These values are invalid, because, upon process failure, FT-MPI destroys all MPI objects with non-local information (e.g., communicators and groups) including its current address space, except MPI_COMM_WORLD, requiring the application to manually recreate these objects after every failure in the same order [9]. So, a flag called *any_worker_re-spawned* is set in the algorithm shown in Fig. 4 on Line 4 to mark that the value in received buffer is invalid. Whether the value is received from restarted processes or not is checked on Line 3 in the algorithm shown in Fig. 4.

- Calculation of *left* and *right* ghost values for workers is done on Line 19 (details in the algorithm shown in Fig. 5). This calculation depends on the value of the flag *any_worker_re-spawned* stated in the algorithm shown in Fig. 4. If that flag is set, then we have to load the saved buffer values in previous iteration (done on Line 9 in the algorithm shown in Fig. 5) into the current buffer before calculating ghost values. Saving the values in current buffer and calculating the ghost values are done on Lines 5 and 6, respectively, in the algorithm shown in Fig. 5. The purpose of Lines 10–15 of the algorithm shown in Fig. 5 is to reset the flag *any_worker_re-spawned* and decrease the main iteration by one, if the flag was set before, so that the algorithm could continue for the correct number of iterations.
- The master is sending advection values including *left* and *right* ghost values as a message to each of the corresponding worker on Line 20 (details in the algorithm shown in Fig. 6) to update their ghost values and replace the buffer with the advection values if it is just rebuilt after failure.
- Sending advection values to master from each worker, receiving advection values including *left* and *right* ghost values from master to each worker, and calculating flux

```
Function void worker_activity(int process_id)

1: do
2:     worker_is_sending_to_master(process_id);
   while (sending is not SUCCESSFUL);
3: do
4:     worker_is_receiving_from_master(process_id);
   while (receiving is not SUCCESSFUL);
5:
6: calculate_flux_and_update_advection_values(process_id);
```

Figure 7. Workers are sending to and receiving from master.

```
Function void recover(MPI_Comm *com, int *er)

1:  MPI_Comm oldcomm, newcomm;
2:  int rc;
3:  int size, rank;
4:  if (*er == MPI_ERR_OTHER) then
5:      oldcomm = MPI_COMM_WORLD;
6:      newcomm = FT_MPI_CHECK_RECOVER;
7:      /* collective recovery occurs here! */
8:      rc = MPI_Comm_dup (oldcomm, &newcomm);
9:      rc = MPI_Comm_rank (MPI_COMM_WORLD, &rank);
10:     rc = MPI_Comm_size (MPI_COMM_WORLD, &size);
11: else
12:     printf("ERR: Error occured with error code %d\n", *er);
13:     sleep(30);
```

Figure 8. Failure recovery function.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-268-4

as well as updating advection values in each worker is done on Line 22 (details in the algorithm shown in Fig. 7).

- Finally, upon completion of main iteration, each worker sends their computer advection values to master so that master can combine these values to generate complete advection values.

A. Failure Detection

Any failure of processes or other errors in communication in FT-MPI is detected by error code `MPI_ERR_OTHER`. This error code is invoked inside a user-defined error handler function (Fig. 8) which is registered as an error handler in main function (Fig. 3) for detecting errors.

B. Failed Process Recovery

The next task after detecting process failure is to recover these failed processes to reconstruct communicator. So, it actually means recovering MPI environment including its communicator. This is done by substituting failed processes with the new ones by passing the FT-MPI attribute `FT_MPI_CHECK_RECOVER` to the collective function `MPI_Comm_dup` shown in the algorithm shown in Fig. 8.

C. Data Recovery Techniques

As we mentioned before, processes replacing failed processes require the application to manually recreate MPI objects other than MPI_COMM_WORLD and needs to initialize the variables in the new address space. There are two following scenarios of process failures on which data recovery technique depends.

- Sending from master is failed.
- Sending from worker is failed.

For the first case, master resends its current buffer to the worker waiting for re-receiving that after recovering from failure. The received data from master after the recovery is valid, because master's address space is not changed due to the worker's failure. The `Do···While` loops of Figs. 6 and 7 are used for resending and re-receiving the buffer. Data recovery techniques under this scenario is demonstrated in the algorithm shown in Fig. 9.

For the second case, on the other hand, worker resends its current buffer after recovering from failure to the master waiting for re-receiving the data of that buffer. The `Do···While` loops of Figs. 4 and 7 are used for re-receiving and resending the buffer. However, the received data from worker after the recovery is invalid to be used as advection values as well as ghost values, because the worker's address space is changed due to its failure as we mentioned before. As a result, a technique should be applied to replace this invalid data into valid one. The technique that we applied is saving the data (advection values) into another buffer in each time-step so that it can be used to replace that invalid

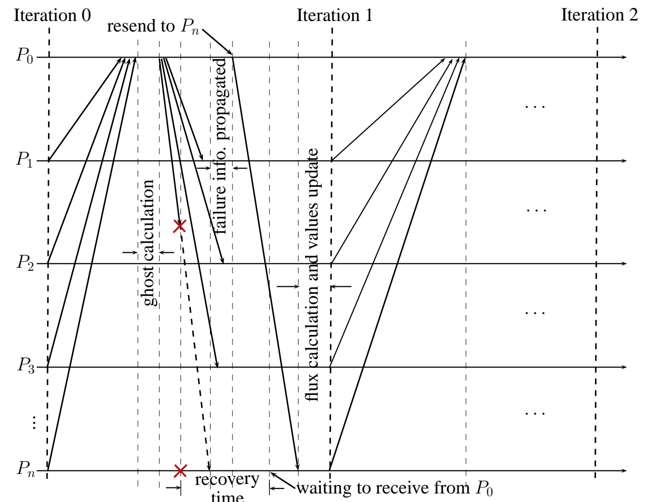

Figure 9. When sending from master is failed.

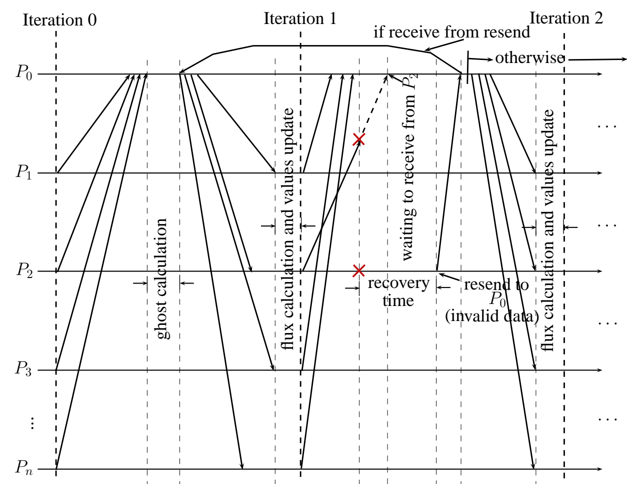

Figure 10. When sending from worker is failed.

buffer. This saving is done on Line 9 in the algorithm shown in Fig. 5. Later, under this scenario, the invalid current buffer is replaced with the saved buffer using Line 5 of the algorithm shown in Fig. 5. Data recovery techniques under this scenario are demonstrated in the algorithm shown in Fig. 10.

D. Time-Evolving Control

The next task after recovering the data for the restarted processes is to control the total number of updates on advection values of the workers. For the scenario of Fig. 10 described in previous section (Section IV-C), the number of such update is decreased by one for each such failure scenario due to invalid data. In order to keep the total number of updates in the presence of failure as the same as that of without failure, the iteration counter should be decreased by one for each such scenario. Lines 10–15 of the algorithm

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-268-4

shown in Fig. 5 is used for this purpose.

E. Duplicate Message Handling

Duplicate message handling in application development using FT-MPI is also an important issue. For the scenario of Figs. 9 and 10, a master or worker does not receive duplicate messages from each other. A re-receive is done only when the previous receive did not succeeded. So, no control is needed for duplicate message handling except `Do···While` loop for resending and re-receiving. However, we should control the receive and send operations on Lines 11 and 13, respectively, of the algorithm shown in Fig. 3 in case of re-spawned process initialization. These two operations are not needed for re-spawned processes. Although there is no problem for send operation in this case as multiple sending operations without corresponding receive operations do not complain, but we should strictly control receive operations. Otherwise, the application waits forever for receiving from master where master sends nothing. An attribute of FT-MPI called `MPI_INIT_RESTARTED_NODE` is used for the purpose of this control.

V. Experimental Result

Although FT-MPI is build with MPI-1 implementation, a significant amount of effort goes into making it competitive with other open source implementations by considering their execution time. In order to prove this issue, we perform an experiment for the non-fault-tolerant (Open MPI) and the Fault-Tolerant (FT-MPI) version of the algorithm applying for the problem discussed in Section IV. This experiment is done on a cluster with a standard GigE Switch with four nodes, each with AMD Phenom(tm) II X4 945 Quad-Core Processor with 3.0GHz of speed and 4.0GB of memory, having a total of 16 cores. The way of measuring the execution time is the `time` and `difftime` functions of `C++`. The result of this experiment is shown in Table I, which shows that FT-MPI is almost similar to Open MPI (version 1.4.5) in case of considering execution time.

Experiment on process failure recovery and recovery time is also conducted for the same problem and on the same cluster, where process failure is simulated by killing the process(es) by issuing the `kill` command and time is measured by the `time` and `difftime` functions of `C++` as before. The experimental result which is performed on a grid with 120 points and 300 time steps is shown in Table II. This result shows that this algorithm could recover from any number of worker process failures. Moreover, the recovery time of process failure is minimum and acceptable.

TABLE I. Execution time of non-fault-tolerant version of algorithm in Open MPI and FT-MPI.

# Grid Points	# Time Steps	Open MPI (Sec)	FT-MPI (Sec)
15360	38400	41	61
30720	76800	173	204
46080	115200	382	441

TABLE II. Experiment on process failure recovery and recovery time.

Total Process Failed	List of Killed Processes	Failure Recovered?	Recovery Time (Sec)
1	Any 1 of the 15 worker processes	YES	1
2	Any 2 of the 15 worker processes	YES	1
3	Any 3 of the 15 worker processes	YES	2
4	Any 4 of the 15 worker processes	YES	2
5	Any 5 of the 15 worker processes	YES	2
6	Any 6 of the 15 worker processes	YES	3
7	Any 7 of the 15 worker processes	YES	3
8	Any 8 of the 15 worker processes	YES	3
9	Any 9 of the 15 worker processes	YES	3
10	Any 10 of the 15 worker processes	YES	4
11	Any 11 of the 15 worker processes	YES	4
12	Any 12 of the 15 worker processes	YES	4
13	Any 13 of the 15 worker processes	YES	5
14	Any 14 of the 15 worker processes	YES	5
15	All worker processes	YES	5

VI. Conclusion

This paper proposes a master-worker model for designing and implementing a fault-tolerant algorithm applicable for time-evolving problems. One of the emerging problems in such category is the solution of advection equations which are modeled by partial differential equations. We have applied this model on the iterative solution of one dimensional advection equations so that it can survive the failure of all the worker processes in that system. We have used the semantics of FT-MPI to implement this algorithm focusing on different issues related to fault tolerance like failure detection, failed process recovery, data recovery techniques, time-evolving control, duplicate message handling, etc. This model is not scalable, but it can recover the failure of all workers in the system and there are scopes to modify this model to make it scalable. This contribution will also help application developers to resolve different issues of design and implementation of fault-tolerant algorithms for more complex time-evolving applications.

We are currently working on modifying this model so that it turns into a scalable solution. One of the scopes include using an extra process for each working process replacing single master process so that each working process can

communicate with another working process directly and save their data to the corresponding extra processes. The purpose of this saving is that the extra processes can send their saved data to the corresponding processes when they re-spawned after the failure. Another approach avoids requiring a master process and saves the data on their *left* and *right* neighbors during communication. This saved data can be send to its neighbors when they are re-spawned after the failure. There are also many approaches like this which can be proposed to make this fault-tolerant application scalable.

ACKNOWLEDGMENT

This work was supported by the Australian Research Council (ARC) and Fujitsu Laboratories of Europe (FLE) through the ARC National Competitive Grants Program (NCGP) Linkage Project LP110200410.

REFERENCES

[1] Y. Ajima, S. Sumimoto, and T. Shimizu, "Tofu: A 6d mesh/torus interconnect for exascale computers," Computer, vol. 42, no. 11, November 2009, pp. 36–40.

[2] L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik, An Introduction to High-Performance Scientific Computing, ser. Scientific and Engineering Computation. MIT Press, 1996.

[3] B. Schroeder and G. A. Gibson, "A large-scale study of failures in high-performance computing systems," in Proc. International Conference on Dependable Systems and Networks, ser. DSN '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 249–258.

[4] A. Geist and C. Engelmann, "Development of naturally fault tolerant algorithms for computing on 100,000 processors," 2002. [Retrieved: December 10, 2012], URL: http://www.csm.ornl.gov/~geist/Lyon2002-geist.pdf

[5] G. Gibson, B. Schroeder, and J. Digney, "Failure tolerance in petascale computers," Software Enabling Technologies for Petascale Science, vol. 3, no. 4, November 2007, pp. 4–10.

[6] Message Passing Interface Forum, "MPI: A message passing interface," in Proc. Supercomputing. IEEE Computer Society Press, November 1993, pp. 878–883.

[7] G. E. Fagg and J. J. Dongarra, "FT-MPI: Fault tolerant mpi, supporting dynamic applications in a dynamic world," 2000.

[8] W. Gropp and E. Lusk, "Fault tolerance in mpi programs," Special issue of the International Journal High Performance Computing Applications (IJHPCA), vol. 18, 2002, pp. 363–372.

[9] J. Hursey and R. Graham, "Building a fault tolerant mpi application: A ring communication example," in IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum (IPDPSW), May 2011, pp. 1549–1556.

[10] Fault Tolerance Working Group, "Run-through stabilization interfaces and semantics." [Retrieved: December 10, 2012], URL: svn.mpi-forum.org/trac/mpi-forum-web/wiki/ft/run_through_stabilization

[11] G. Bosilca, R. Delmas, J. Dongarra, and J. Langou, "Algorithm-based fault tolerance applied to high performance computing," Journal of Parallel and Distributed Computing, vol. 69, no. 4, 2009, pp. 410–416.

[12] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," IEEE Trans. Comput., vol. 33, no. 6, June 1984, pp. 518–528.

[13] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," SIGPLAN Not., vol. 47, no. 8, February 2012, pp. 225–234.

[14] 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 Proc. tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPoPP '05. New York, NY, USA: ACM, 2005, pp. 213–223.

[15] J. Langou, Z. Chen, G. Bosilca, and J. Dongarra, "Recovery patterns for iterative methods in a parallel unstable environment," SIAM J. Sci. Comput., vol. 30, no. 1, November 2007, pp. 102–116.

[16] Z. Chen, "Algorithm-based recovery for iterative methods without checkpointing," in Proc. 20th International Symposium on High Performance Distributed Computing, ser. HPDC '11. New York, NY, USA: ACM, 2011, pp. 73–84.

[17] G. E. Fagg and J. Dongarra, "Building and using a fault-tolerant mpi implementation," vol. 18, no. 3, 2004, pp. 353–361.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-268-4