Volume 8, No. 3, March – April 2017

ISSN No. 0976-5697

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

Fault Tolerance and Message Passing Interface Programs

Dr. Mohammad Miyan
Associate Professor, Shia P. G. College, University of Lucknow
Sitapur Road, Lucknow, India

Abstract: In this paper we have mentioned the means that of fault tolerance as a program property that ensures survival of adequate state for continuing the program. We've got surveyed of various researches, use cases and example programs what the MPI customary provides among the approach of support for writing fault-tolerant programs. We've got thought of the many approaches to doing this, which we've incontestable but one can write fault-tolerant MPI programs. We conclude that, inside bound constraints, MPI will offer a helpful context for writing application programs that exhibit important degrees of fault tolerance.

Keywords: Checkpointing; Fault tolerance; Implementation; MPI, Process management.

I. INTRODUCTION

The fault tolerance is that the property that allows a system to continue operational properly within the event of the failure of (or one or a lot of faults within) a number of its parts. If its operational quality decreases the least bit, the decrease is proportional to the severity of the failure, as compared to a naively designed system within which even a tiny low failure will cause total breakdown. Fault tolerance is especially wanted in high-availability or life-critical systems. The power of maintaining practically once parts of a system break down is observed as sleek degradation [1]. A fault-tolerant style allows a system to continue its supposed operation, presumably at a reduced level, instead of failing fully, once some a part of the system fails. The term is most ordinarily accustomed describe computer systems designed to continue a lot of or less absolutely operational with, perhaps, a discount in output discount in output or a rise in time interval within the event of some partial failure. That is, the system as a full isn't stopped because of issues either within the hardware or the package software. Associate example in another field may be a motorcar designed thus it'll still be drivable if one among the tires is pierced. A structure is in a position to retain its integrity within the presence of harm because of causes like fatigue, corrosion, producing flaws, corrosion, producing flaws, or impact [1].

Within the scope of a private system, fault tolerance will be achieved by anticipating exceptional conditions and building the system to address them, and, in general, aiming for self-stabilization in order that the system converges towards associate in nursing error-free state. However, if the implications of a system failure are harmful, or the value of creating it sufficiently reliable is incredibly high, a far better resolution is also to use some sort of duplication. In any case, if the consequence of a system failure is thus harmful, the system should be able to use reversion to fall back to a secure mode. This will be kind of like roll-back recovery however can be somebody's action if humans exist within the loop. Fault-tolerant computer systems are systems designed round the ideas of fault tolerance. In essence, they need to be able to continue operating to a level of satisfaction within the presence of faults. Fault tolerance isn't simply a property of individual machines; it's going to conjointly characterize the principles by that they move. For instance, the Transmission Control Protocol (TCP) is meant to permit reliable two-way communication in a very packet-switched network, even within the presence of communications links that are imperfect or full. It wills this by requiring the endpoints of the communication to expect packet loss, duplication, reordering, and corruption in order that these conditions don't harm knowledge integrity, and solely scale back output by a proportional quantity [1], [2], [3], [4].

As trendy supercomputers scale to a whole bunch or perhaps thousands of individual nodes, the Message Passing Interface (MPI) remains a simple and effective thanks to program them. At an equivalent time, the larger range of individual hardware elements implies that hardware faults are additional doubtless to occur throughout long running jobs. Users naturally wish their programs to adapt to hardware faults and continue running. This ideal is clearly unachievable normally e.g. if all nodes fail, however users still can do a big degree of fault tolerance for his or her MPI programs [5]. Most MPI implementations incorporates a particular set of routines directly due from C, C++, FORTRAN i.e., API and any language able to interface with such libraries, together with C, Java or Python. The benefits of MPI over older message passing libraries are non-movable as MPI has been enforced for all the distributed memory architecture; and speed since every implementation is in theory optimized for the hardware on that it runs. MPI uses Language Independent Specifications (LIS) for calls and language bindings. The first MPI standard specified ANSI C and FORTRAN-77, which bindings each other with LIS. The draft was bestowed at Supercomputing 1994 and finalized before long thenceforth. Concerning 128 functions represent the MPI-1.3 standard that was output as the final finish of the MPI-1 series in 2008 [5].

At present, the standard has many versions: version 1.3 i.e., said as MPI-1 that emphasizes message passing and encompasses a static runtime setting, MPI-2.2 (MPI-2), which has new options like parallel I/O, dynamic method management and remote memory operations, and MPI-3.1 (MPI-3), which has extensions to the collective operations with non-blocking versions and extensions to the one-sided operations. MPI-2's LIS specifies over five hundred functions and provides language bindings for ISO C, ISO C++, and FORTRAN 90. Object ability was additionally additional to permit easier mixed-language message passing programming. A side-effect of standardizing MPI-2, completed in 1996, was instructive the MPI-1 customary, making the MPI-1.2. MPI-2

is generally a superset of MPI-1, though some functions are deprecated. MPI-1.3 programs still work below MPI implementations compliant with the MPI-2 customary. MPI-3 includes new FORTRAN 2008 bindings, whereas it removes deprecated C++ bindings in addition as several deprecated routines and MPI objects [6].

MPI is commonly compared with Parallel Virtual Machine (PVM), that could be a widespread distributed surroundings and message passing system developed in 1989, and that was one among the systems that actuated the necessity for traditional parallel message passing. Rib shared memory programming models like as Pthreads and OpenMP; and message passing programming i.e., MPI/PVM, is thought of as complementary programming approaches, and might sometimes be seen along in applications, e.g. in servers with multiple massive shared-memory nodes [5]. The MPI interface is supposed to produce essential virtual topology, synchronization, and communication practicality between a collection of processes that are mapped to nodes, servers and computer instances; in a very language-independent manner, with language-specific syntax and many language-specific options. MPI programs forever work with processes; however programmers usually sit down with the processes as processors. Typically, for max performance, every processing unit i.e., C.P.U. or any other unit or core in a very multi-core machine are going to be appointed simply one process. This assignment happens at runtime through the agent that starts the MPI program, commonly referred to as mpirun or mpiexec [7]. MPI library functions embody, however don't seem to be restricted to, point-to-point rendezvous-type send/receive operations, selecting between a Cartesian or graph-like logical method topology, exchanging knowledge between method pairs i.e., send/receive operations, combining partial results of computations i.e., gather and cut back operations, synchronizing nodes i.e., barrier operation moreover as getting network-related data like the quantity of processes within the computing session, current processor identity that a process is mapped to, neighboring processes accessible in an exceedingly network topology, and so on. Point-to-point operations are available synchronous, asynchronous, buffered, and prepared forms, to permit stronger and weaker linguistics for the synchronization aspects of a rendezvous- send. Several outstanding operations are doable in asynchronous mode, in most implementations. MPI-1 and MPI-2 each engine implementations that overlap communication and computation, however apply and theory dissent. MPI conjointly specifies thread safe interfaces, that have cohesion and coupling methods that facilitate avoid hidden state inside the interface. It's comparatively simple to write down multithreaded point-to-point MPI code, and a few implementations support such code [5], [6], [7].

II. INTERFACE SPECIFICATION

- MPI primarily addresses the message-passing parallel programming model: information is affected from the address area of one method to it of another method through cooperative operations on every method.
- MPI may be a specification for the developers and users of message passing libraries. By itself, it's NOT a library - however rather the specification of what such a library ought to be.
- Simply expressed, the goal of the Message Passing Interface is to supply a wide used common place for writing message passing programs. The interface tries to be portable, practical, flexible and efficient.
- The MPI standards has well-versed variety of revisions, with the foremost recent version being MPI-3.x.
- Actual MPI library implementations dissent during which version and options of the MPI normal they support. Developers and users ought to bear in mind of this.
- Interface specifications are outlined for C and FORTRAN90 language bindings. The MPI-3 additionally provides support for FORTRAN 2003 and 2008 options and C++ bindings from MPI-1 are diminished in MPI-3 [6].

III. PROGRAMMING MODEL

The programming model is as follows [6]:

- Originally, MPI was constructed for distributed memory architectures that were changing into progressively widespread at that point (1980 - 1990) as shown in figure 1.

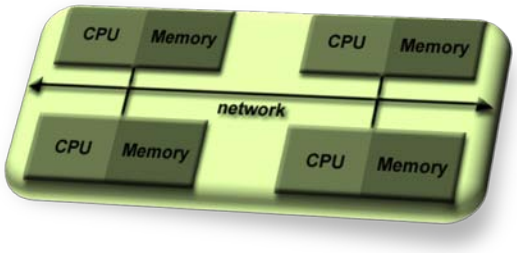

Figure 1

- As design trends modified, shared memory SMPs were combined over networks making hybrid distributed memory /shared memory systems.
- MPI implementers custom-made their libraries to handle each sorts of underlying memory architectures seamlessly. They additionally developed ways in which of handling totally different interconnects and protocols as shown by figure 2.

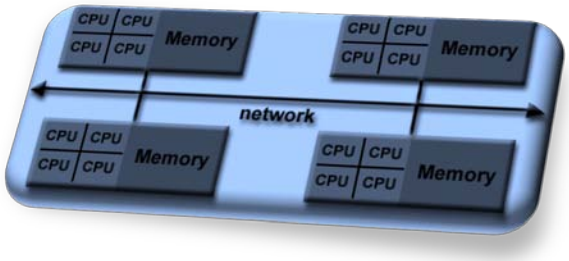

Figure 2

- The programming model clearly remains a distributed memory model but, notwithstanding the underlying physical design of the machine. Today, MPI runs on just about any hardware platform i.e., Distributed Memory, Shared Memory and Hybrid.
- All similarity is explicit: the computer programmer is accountable for properly distinctive similarity and implementing parallel algorithms victimization MPI constructs.

IV. MPI USAGES

The MPI usages are as given [8]:

- Standardization - MPI is that the solely message passing library that may be thought of a customary. It's supported on nearly all HPC platforms. Much, it's replaced all previous message passing libraries.
- Functionality - There are over 430 routines outlined in MPI-3, which incorporates the bulk of these in MPI-2 and MPI-1. Most MPI programs are often written employing a dozen or less routines.
- Performance Opportunities - merchant implementations ought to be able to exploit native hardware options to optimize performance. Any implementation is unengaged to develop optimized algorithms.
- Availability - a spread of implementations are obtainable, each merchant and property right.

V. RELATED RESEARCHES

G. E. Fagg et al., (2001) [9], have given an overview of the FT-MPI semantics, design, applications, tools and the performance. They also discussed about the experiment HARNESS core implementation, which FT-MPI is built for operation.

R. Batchu et al., (2001) [10], have presented a fault-tolerant methodology leading to new MPI implementations, which provides the support for successful completion of MPI applications in the presence of random, transient faults, recurring, induced extraneously.

G. E. Fagg et al., (2004) [11], have discussed the design and uses of a fault-tolerant MPI, which handles the process failures in a way beyond that of the original MPI static process model. The FTMPI allows the semantics and related modes of failures that were explicitly controlled by an application through a modified functionality within standard MPI 1.2 API.

R. Batchu et al., (2004) [12], have described the design and implementation of MPI/FT, a high-performance MPI-1.2 implementation enhanced with low overhead functionality to detect and recover from process failures.

P. Balaji et al., (2013) [13], have presented the one sided communication and two sided communication models as shown in figure-3 and figure-4 respectively [13].

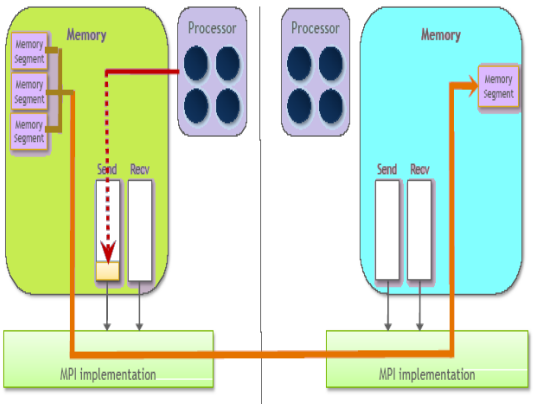

Figure 3 One sided communication model

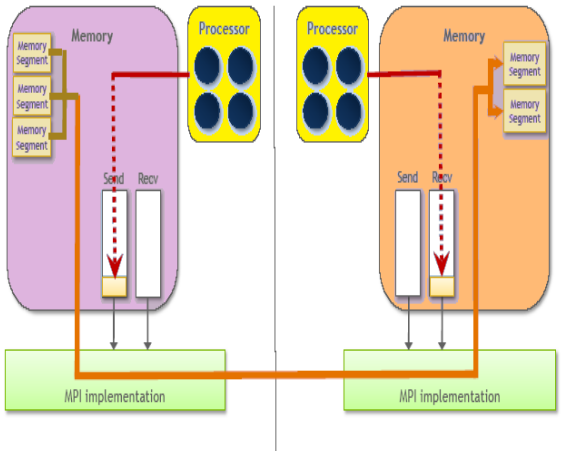

Figure 4 Two sided communication model

I. Lagunay et al., (2016) [14], have investigated a different model for MPI fault tolerance i.e., a global-exception, roll-back recovery model. In contrast to ULFM, the basic idea of the model was that upon detecting a fail-stop failure, MPI reinitializes itself, it returns MPI to its state prior to returning from MPI_Init, and so that restarts application at an application specified restart point.

VI. USE CASES AND EXAMPLES

The some common use cases and examples [15] are as follows:

A. Transparent Checkpoint to NFS

This use case demonstrates the essential checkpoint/restart practicality of Open MPI. Checkpoints are keeping on to a globally mounted filing system in the/home/me/checkpoints/ directory. For this instance we have a tendency to assume application without modifying and Open MPI victimization the BLCR library for generating the local snapshots.

$HOME/.openmpi/mca-params.conf
 # Local snapshot directory (not used in this scenario)
 # crs_base_snapshot_dir was deprecated in r23587, and in v1.5.1 and later releases.
 # crs_base_snapshot_dir=/home/me/tmp
sstore_stage_local_snapshot_dir=/home/me/tmp
 # Remote snapshot directory (globally mounted file system)
 # snapc_base_global_snapshot_dir was deprecated in r23587, and in v1.5.1 and later releases.
 # snapc_base_global_snapshot_dir=/home/me/checkpoints
sstore_base_global_snapshot_dir=/home/me/checkpoints

a) Shell #1:

Start an MPI job enabling fault tolerance. (Assume that the PID of mpirun is 1234).

shell$ mpirun -am ft-enable-cr my-app <args>
...

b) Shell #2:

Checkpoint the MPI job with mpirun PID 1234. The second checkpoint, terminate the job.

shell$ ompi-checkpoint 1234
Snapshot Ref.: 0 ompi_global_snapshot_1234.ckpt
shell$ echo "wait for some time..."
shell$ ompi-checkpoint --term 1234
Snapshot Ref.: 1 ompi_global_snapshot_1234.ckpt
shell$

c) Shell #1:

Restart the job from the most recent checkpoint

shell$ ompi-restart ompi_global_snapshot_1234.ckpt
...

B. Transparent Checkpoint to Local Disk

This use case demonstrates the basic checkpoint/restart functionality of Open MPI. Checkpoints are stored directly to a globally mounted file system in the/home/ me /checkpoints/ directory. For this example we assume an unmodified application and Open MPI using the BLCR library for generating local snapshots.

1) $HOME/.openmpi/mca-params.conf

```
# Transfer the files from the local snapshot directory to the global snapshot
# directory
# snapc_base_store_in_place was deprecated in r23587, and in v1.5.1 and later releases.
# snapc_base_store_in_place=0
sstore=stage
# Local snapshot directory (locally mounted file system)
# crs_base_snapshot_dir was deprecated in r23587, and in v1.5.1 and later releases.
# crs_base_snapshot_dir=/tmp/me/local
sstore_stage_local_snapshot_dir=/tmp/me/local
# Remote snapshot directory (locally mounted file system))
# snapc_base_global_snapshot_dir was deprecated in r23587, and in v1.5.1 and later releases.
# snapc_base_global_snapshot_dir=/tmp/me/global
sstore_base_global_snapshot_dir=/tmp/me/global
```

a) Shell #1:

Start an MPI job enabling fault tolerance. (Assume that the PID of mpirun is 1234).

shell$ mpirun -am ft-enable-cr my-app <args>
...

b) Shell #2:

Checkpoint the MPI job with mpirun PID 1234. The second checkpoint, terminate the job.

```
shell$ ompi-checkpoint 1234
Snapshot Ref.: 0 ompi_global_snapshot_1234.ckpt
shell$ echo "wait for some time..."
shell$ ompi-checkpoint --term 1234
Snapshot Ref.: 1 ompi_global_snapshot_1234.ckpt
shell$
```

c) Shell #1:

Restart the job from most recent checkpoint. Make sure to pass the --preload option, so the checkpoint files are transferred to the remote system during startup.

shell$ ompi-restart --preload ompi_global_snapshot_1234.ckpt
...

C. *Checkpointing and SIGSTOP/SIGCONT*

This use case demonstrates the way to stop associate degreed instantly send SIGSTOP to an Open MPI application. The applying will then be continued through SIGCONT. Instead the applying may be terminated, and restarted at a later purpose in time from the generated checkpoint. This practicality is helpful in an exceedingly gang regular atmosphere wherever a running application could also be stopped and command in memory whereas another application uses the machines. The new application will safely kill the stopped application if it desires additional memory, since the stopped application is restarted from a stop. Instead if additional resources become accessible the stopped application is terminated and restarted on the free resources.

a) Shell #1:

Start an MPI job enabling fault tolerance. (Assume that the PID of mpirun is 1234).

shell$ mpirun -am ft-enable-cr my-app <args>
...

b) Shell #2:

Checkpoint the MPI job with mpirun PID 1234 passing the --stop option to send SIGSTOP to application just after the checkpointing. The checkpoint generated can be used as usual. If restarting the SIGCONT signal is automatically forwarded to the restarted processes.

```
shell$ ompi-checkpoint --stop -v 1234
[localhost:001300] [  0.00 /  0.20]          Requested - ...
[localhost:001300] [  0.00 /  0.20]           Pending - ...
[localhost:001300] [  0.01 /  0.21]           Running - ...
[localhost:001300] [  1.01 /  1.22]                 Stopped - ompi_global_snapshot_1234.ckpt
Snapshot Ref.: 0 ompi_global_snapshot_1234.ckpt
shell$ echo "Application is now stopped"
shell$
```

c) Shell #2:

To resume the job just send the SIGCONT signal to mpirun. That will forward the signal to all of the processes in the application.

```
shell$ kill -CONT 1234
shell$ echo "Application resumes computation"
```

D. SELF Checkpoint/Restart System

The SELF component can invoke the user-defined functions to avoid wasting and restore checkpoints. It's merely a mechanism for a user-defined operates to be invoked at Open MPI's stop, Continue, and Restart phases. Hence, the sole information that's saved throughout the stop is what's written within the users stop operate - no MPI library state is saved in any respect. As such, the model for the SELF-component is slightly completely different than, as an example, the BLCR part. Specifically, the Restart operate isn't invoked within the same method image of the method that was check pointed. The Restart part is invoked throughout MPI_INIT of a brand new instance of the applying i.e., it starts over from main(). Below is associate example of associate application that takes advantage of the SELF Checkpoint/Restart System. Stopping and restarting of the MPI job happens specifically as within the clear Checkpoint Use Cases.

a) Compiling

mpicc my-app.c -export -export-dynamic -o my-app

b) Running

shell$ mpirun -np 2 -am ft-enable-cr my-app

shell$ mpirun -np 2 -am ft-enable-cr -mca crs_self_prefix my_personal my-app

c) my-app.c:

```
/*
 * Example Open PAL CRS self program
 * Author: Joshua Hursey
 */
#include <mpi.h>
#include <stdio.h>
#include <signal.h>
#include <string.h>
#define LIMIT 100
```

```
/*************************
 * Function Declarations
 *************************/
void signal_handler(int sig);
/* Default OPAL crs self callback functions */
int opal_crs_self_user_checkpoint(char **restart_cmd);
int opal_crs_self_user_continue(void);
int opal_crs_self_user_restart(void);
/* OPAL crs self callback functions */
int my_personal_checkpoint(char **restart_cmd);
int my_personal_continue(void);
int my_personal_restart(void);
/********************
 * Global Variables
 ********************/
int am_done = 1;
int current_step  = 0;
char ckpt_file[128] = "my-personal-cr-file.ckpt";
char restart_path[128] = "/full/path/to/personal-cr";
/**********
 *  Main
 **********/
int main(int argc, char *argv[]) {
   int rank, size;
      current_step = 0;
   MPI_Init(&argc, &argv);
   /* So we can exit cleanly */
   signal(SIGINT,  signal_handler);
   signal(SIGTERM, signal_handler);
   for(; current_step <  LIMIT; current_step += 1) {
      printf("%d) Step %d\n", getpid(), current_step);
      sleep(1);
      if(0 == am_done) {
         break;
      }
   }
   MPI_Finalize();
   return 0;
}
void signal_handler(int sig) {
   printf("Received Signal %d\n", sig);
   am_done = 0;
}
/* OPAL crs self callbacks for checkpoint */
int opal_crs_self_user_checkpoint(char **restart_cmd) {
   printf("opal_crs_self_user_checkpoint callback...\n");
   my_personal_checkpoint(restart_cmd);
   return 0;
}
int opal_crs_self_user_continue(void) {
   printf("opal_crs_self_user_continue callback...\n");
   my_personal_continue();
   return 0;
}
int opal_crs_self_user_restart(void) {
   printf("opal_crs_self_user_restart callback...\n");
   my_personal_restart();
   return 0;
}
/* OPAL crs self callback for checkpoint */
int my_personal_checkpoint(char **restart_cmd) {
   FILE *fp;
      *restart_cmd = NULL;
   printf("my_personal_checkpoint callback...\n");
      /*
    * Open our checkpoint file
    */
   if( NULL == (fp = fopen(ckpt_file, "w")) ) {
      fprintf(stderr, "Error: Unable to open file (%s)\n",
ckpt_file);
      return;
   }
      /*
    * Save the process state
    */
   fprintf(fp, "%d\n", current_step);
      /*
    * Close the checkpoint file
    */
   fclose(fp);
   /*
    * Figure out the restart command
    */
   asprintf(restart_cmd, "%s", strdup(restart_path));
   return 0;
}
int my_personal_continue() {
   printf("my_personal_continue callback...\n");
   /* Don't need to do anything here since we are in the
    *  state that we want to be in already.
    */
   return 0;
}
int my_personal_restart() {
   FILE *fp;
   printf("my_personal_restart callback...\n");
   /*
    * Open our checkpoint file
    */
   if( NULL == (fp = fopen(ckpt_file, "r")) ) {
      fprintf(stderr, "Error: Unable to open file (%s)\n",
ckpt_file);
      return;
   }
   /*
    * Access the process state that we saved and
    * update the current step variable.
    */
   fscanf(fp, "%d", ¤t_step);

   fclose(fp);
   printf("my_personal_restart: Restarting from step
%d\n", current_step);
      return 0;
}
```

Here could also be a "Hello World" program in MPI written in C [16]. During this example, there is tending to tend to send a "hello" message to each processor, manipulate it trivially, results to the fore most technique, and print the messages.

```
/*
  "Hello World" MPI Test Program
*/
#include <assert.h>
#include <stdio.h>
#include <string.h>
```

```
#include <mpi.h>
int main(int argc, char **argv)
{
   char buf[256];
   int my_rank, num_procs;
   /* Initialize the infrastructure necessary for
communication */
   MPI_Init(&argc, &argv);
   /* Identify this process */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   /* Find out how many total processes are active */
   MPI_Comm_size(MPI_COMM_WORLD,
&num_procs);
   /* Until this point, all programs have been doing
exactly the same.
      Here, we check the rank to distinguish the roles of
the programs */
   if (my_rank == 0) {
      int other_rank;
      printf("We have %i processes.\n", num_procs);
      /* Send messages to all other processes */
      for (other_rank = 1; other_rank < num_procs;
other_rank++)
      {
         sprintf(buf, "Hello %i!", other_rank);
         MPI_Send(buf, sizeof(buf), MPI_CHAR,
other_rank,
               0, MPI_COMM_WORLD);
      }
      /* Receive messages from all other process */
      for (other_rank = 1; other_rank < num_procs;
other_rank++)
      {
         MPI_Recv(buf, sizeof(buf), MPI_CHAR,
other_rank,
               0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
         printf("%s\n", buf);
      }
   } else {
      /* Receive message from process #0 */
      MPI_Recv(buf, sizeof(buf), MPI_CHAR, 0,
            0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      assert(memcmp(buf, "Hello ", 6) == 0),
      /* Send message to process #0 */
      sprintf(buf, "Process %i reporting for duty.",
my_rank);
      MPI_Send(buf, sizeof(buf), MPI_CHAR, 0,
            0, MPI_COMM_WORLD);
   }
   /* Tear down the communication infrastructure */
   MPI_Finalize();
   return 0;
}
```

When the program is running with 4 processes, it will give the output:

$ mpicc example.c && mpiexec -n 4 ./a.out

There are 4 processes i.e., process 1 reporting for duty, process 2 reporting for duty, process 3 reporting for duty. Here, mpiexec could also be a command accustomed execute the instance program with four processes, each of that's associate degree freelance instance of the program at the run time and assigned ranks i.e., numeric 0, 1, 2, and 3. The name mpiexec is sometimes counseled by the MPI customary, though some implementations provide a regular command below the name mpirun. The MPI_COMM_WORLD is that the mortal that relates to any or all the processes [16].

VII. CHECKPOINTING

Checkpointing could be a technique to feature fault tolerance into computing systems. It primarily consists of saving an exposure of the application's state, so it will restart from that time just in case of failure. This can be notably necessary for long-running applications that are dead in failure-prone computing systems. In distributed computing, checkpointing could be a technique that helps tolerate failures that otherwise would force long-running application to restart from the start. The foremost basic thanks to implement checkpointing is to prevent the appliance, copy all the desired information from the memory to reliable storage e.g., parallel file system; and so continue with the execution. Within the case of failure, once the appliance restarts, it doesn't have to be compelled to begin from scratch. Rather, it'll scan the most recent state "the checkpoint" from the stable storage and execute from that.

There are two important approaches for checkpointing in such systems i.e., coordinated checkpointing and uncoordinated checkpointing. Within the coordinated checkpointing approach, processes should make sure that their checkpoints square measure consistent. This can be sometimes achieved by some reasonably two-phase commit protocol algorithmic rule. In uncoordinated checkpointing, every method checkpoints is its own state independently. It should be stressed that merely forcing processes to stop their state at fastened time intervals isn't spare to make sure world consistency. The requirement for establishing a standardized state i.e., no missing messages or duplicated messages might force alternative processes to roll back to their checkpoints, that successively might cause alternative processes to roll back to even earlier checkpoints, that within the most extreme case might mean that the sole consistent state found is that the initial state i.e., the alleged domino effect [17].

VIII. APPLICATION IMPLEMENTATIONS

A. Save State

One of the initial and currently commonest means that of application checkpointing was a "save state" feature in interactive applications, during which the user of the applying may save the state of all variables and different information to a data-storage medium at the time they were victimization it and either continue operating, or exist the applying and at a later time, restart the applying and restore the saved state. This was enforced through a "save" command or menu choice within the application. In several cases, it became common place apply to raise the user if that they had cursed work once exiting the applying if they wished to save lots of their work before doing therefore. This sort of practicality became very vital for usability in applications wherever the actual work couldn't be completed in one sitting such as taking part in a computer game expected to require dozens of hours, or writing a book or long document amounting to a whole lot or thousands of pages or wherever the work was being done over an extended amount of your time like information entry into a

document like rows in an exceedingly computer program. The problem with save state is it needs the operator of a program to request the save. For non-interactive programs, together with machine-controlled or batch processed workloads, the flexibility to stop such applications conjointly had to be machine-controlled [17].

B. Checkpoint/ Restart

As batch applications began to handle tens to many Thousands of group actions wherever every transaction would possibly method one record from one against file many totally different files the necessity for the applying to be restart at some purpose while not the necessity to rerun the whole job from scratch became imperative. So the "checkpoint/restart" capability was born, during which when variety of transactions had been processed, a "snapshot" or "checkpoint" of the state of the applying may be taken, at that purpose if the applying failing before future stop it may be restarted by giving it the stop data and therefore the last place within the detail file wherever a group action had with success completed. The applying might then restart at that time.
Checkpointing would tend to be dearly-won, therefore it absolutely was typically not finished each record, however at some affordable compromise between the checkpoint and the value of the computer time required to utilize a batch of records. So the quantity of records processed for every stop would possibly vary from twenty five to two hundred, counting on cost factors and therefore the relative quality of the applying and therefore the resources required to with success restart the applying [17].

C. Fault Tolerance Interface (FTI)

FTI may be a library that aims to produce machine scientists with simple thanks to perform checkpoint/restart in a very ascendible fashion. FTI leverages native storage and multiple replications and erasures techniques to produce many levels of responsibility and performance. FTI provides application-level checkpointing that enables users to pick that information has to be protected, so as to boost potency and avoid house, time and energy waste. It offers an instantaneous information interface in order that users don't have to cope with files and/or directory names. All information is managed by FTI in a very clear fashion for the user. If desired, users will dedicate one method per node to overlap fault tolerance work and scientific computation, in order that post-checkpoint tasks are executed asynchronously [17].

D. Berkeley Lab Checkpoint/Restart (BLCR)

The Future Technologies cluster at the Lawrence National Laboratories is developing a hybrid kernel/user implementation of checkpoint/restart referred to as BLCR. Their goal is to produce a strong, production quality implementation that checkpoints a good vary of applications, while not requiring changes to be created to application code. BLCR focuses on checkpointing parallel applications that communicate through MPI, and on compatibility with the computer software created by the SciDAC scalable computer program ISIC. Its work is lessened into four main areas: Checkpoint/Restart for UNIX operating system (CR), Checkpoint able MPI Libraries, Resource Management Interface to Checkpoint/Restart and Development of method Management Interfaces [17].

E. DMTCP

DMTCP (Distributed Multithreaded Checkpointing) may be a tool for transparently checkpointing the state of discretionary cluster of programs unfold across several machines and connected by sockets. It doesn't modify the user's program or the software system. Among the applications supported by DMTCP are Open MPI, Python, Perl, and plenty of programming languages and shell scripting languages. With the employment of TightVNC, it may also stop and restart X Window applications, as long as they are doing not use extensions e.g. any OpenGL or video. Among the UNIX operating system options supported by DMTCP are open file descriptors, pipes, sockets, signal handlers, method id and thread id virtualization i.e., ensure recent pids and tids still work upon restart, ptys, fifos, method cluster ids, session ids, terminal attributes, and mmap/mprotect including mmap-based shared memory. DMTCP supports the OFED API for Infini Band on experimental basis [17].

F. Collaborative Checkpointing

Some recent protocols perform cooperative stopping by storing fragments of the checkpoint in close nodes. This can be useful as a result of it avoids the price of storing to a parallel filing system, which often becomes a bottleneck for large-scale systems and it uses storage that's nearer. This has found use notably in large-scale supercomputing clusters. The challenge is to confirm that once the stop is required once sick from a failure, the close nodes with fragments of the checkpoints are on the market [17].

G. Docker

Docker and therefore the underlying technology contain a stop and restore mechanism.

H. CRIU

CRIU could be a user area checkpoint library.

IX. EMBEDDED AND ASIC DEVICES IMPLEMENTATIONS

A. Mementos

Mementos could be a code that remodels all-purpose tasks into interruptible program for platforms with frequent power outages. It's been designed for battery less embedded devices like RFID tags and good cards that have confidence harvest home energy from close background. Mementos like senses the on the market energy within the system, and decides to stop the program or continue the computation. Just in case of checkpointing, information is hold on in an exceedingly non-volatile memory. Once the energy become adequate for revive, the info are retrieved from the memory, and therefore the program continues from the hold on state. Mementos have been enforced on the MSP430 family of microcontrollers. Souvenirs are called once patron saint Nolan's Memento.

B. Idetic

Idetic could be a set of automatic tools that helps Application-specific computer circuit (ASIC) developers to mechanically enter checkpoints in their styles. It targets high-level synthesis tools and adds the checkpoints at the register-transfer level i.e., Verilog code. It uses a dynamic programming approach to find low overhead points within the state machine of the planning. Since the checkpointing in hardware level involves

causing the info of dependent registers to a non-volatile memory, the optimum points are it needed to own minimum variety of registers to store. Idetic is deployed and evaluated on energy harvest home RFID tag device [17].

X. CONCLUSIONS

In this paper we've got mentioned the means of fault tolerance as a program property that ensures survival of adequate state for continued the program. We've got surveyed what the MPI customary provides within the approach of support for writing fault-tolerant programs. We've got thought of many approaches to doing this, and that we have incontestable however one will write fault-tolerant MPI programs.

XI. REFERENCES

[1] Wikipedea, Free encyclopidea on Fault Tolerance, 2017. https://en.wikipedia.org/wiki/Fault_tolerance.

[2] B. W. Johnson, "Fault-Tolerant Microprocessor-Based Systems", IEEE Micro, vol. 4, no. 6, 1984, pp. 6-21

[3] D. K. Pradhan, "Fault-tolerant computer system design book contents," ISBN 0-13-057887-8, pp. 135 - 235, 1996.

[4] J. Vytopil, "Formal Techniques in Real-Time and Fault-Tolerant Systems," Second International Symposium, Nijmegen, the Netherlands, January 8–10, 1992, Proceedings Published by Springer, 1991, ISBN 3-540-55092-5, 978-3-540-55092-1.

[5] Wikipedia, Free Encyclopedia on Message Passing Interface, 2017. https://en.wikipedia.org/wiki/Message_Passing_Interface

[6] G. William, L. Ewing and S. Anthony, "Using MPI, 2nd Edition: Portable Parallel Programming with the Message Passing Interface," Cambridge, MA, USA: MIT Press Scientific and Engineering Computation Series, 1999(a), ISBN 978-0-262-57132-6.

[7] G. William, L. Ewing and S. Anthony, "Using MPI-2: Advanced Features of the Message Passing Interface," MIT Press, 1999(b). ISBN 0-262-57133-1.

[8] B. Barney, "Tutorials onMessage Passing Interface (MPI)," Lawrence Livermore National Laboratory, 2017. https://computing.llnl.gov/tutorials/mpi/

[9] G. E Fagg, A Bukovsky and J. J. Dongarra, "HARNESS and fault tolerent MPI," Parallel Commputing, 27, 2001 pp. 1479-1495.

[10] R. Batchu et al., "MPI/FTTM: Architecture and Taxonomies for Fault-Tolerant, Message-Passing Middleware for Performance-Portable Parallel Computing,"Work performed in part with support from NASA under subcontract, 1219475, from the Jet Propulsion Laboratory, 2001, California Institute of Technology.

[11] G. E. Fagg and J. J. Dongarra, "Building and Using a Fault-Tolerant MPI Implementation," *International Journal of High Performance Computing Applications*, 2004, 18: 353. DOI: 10.1177/1094342004046052

[12] R. Batchu and Y. S. Dandass, "MPI/FT: A Model-Based Approach to Low-Overhead Fault Tolerant Message-Passing Middleware," Cluster Computing 7, 303–315, 2004.

[13] P. Balaji and T. Hoefler, "Advanced Parallel Programming with MPI-1, MPI-2, and MPI-3," PPoPP, Argonne National Lab., Shenzhen, China, 2013.

[14] I. Lagunay et al., "A Global Exception Fault Tolerance Model for MPI," Lawrence Livermore National Laboratory, Los Alamos National Laboratory, New Mexico Institute of Mining and Technology, 2016.

[15] Fault Tolerance Research @ Open Systems Laboratory, 2017. http://www.crest.iu.edu/research/ft/ompi-cr/examples.php

[16] Wikipedia, Message Passing Interface, (free encyclopedia), 2017. https://en.wikipedia.org/wiki/Message_Passing_Interface

[17] Wikipedia, Application Checkpointing, (free encyclopedia), 2017. https://en.wikipedia.org/wiki/Application_checkpointing