Data Persistency for Fault-Tolerance Using MPI Semantics

José Gracia,* Nico Struckmann,* Julian Rilli,†‡ Rainer Keller‡
*High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany
†University of Tübingen, Tübingen, Germany
‡University of Applied Sciences, HfT Stuttgart, Germany
Email: gracia@hlrs.de, struckmann@hlrs.de, julian@rilli.eu, rainer.keller@hft-stuttgart.de

***Abstract*—As the size and complexity of high-performance computing hardware, as well as applications increase, the likelihood of a hardware failure during the execution time of large distributed applications is no longer negligible. On the other hand, frequent checkpointing of full application state or even full compute node memory is prohibitively expensive. Thus, application-level checkpointing of only indispensable data and application state is the only viable option to increase an application's resiliency against faults. Existing application-level checkpointing approaches, however, require the user to learn new programming interfaces, etc. In this paper we present an approach to persist data and application state, as for instance messages transfered between compute nodes, which is seamlessly integrated into Message Passing Interface, i.e., the de-facto standard for distributed parallel computing in high-performance computing. The basic idea consists in allowing the user to mark a given communicator as having special, i.e., persistent, meaning. All communication through this persistent communicator is stored transparently by the system and available for application restart even after a failure. The concept is demonstrated by prototypical implementation of the proposed interface.**

Keywords–Message Passing Interface; MPI; fault-tolerance; application-level checkpointing; data persistency

I. Introduction

This paper builds on top of work presented in [1].

Numerical simulation on high-performance computing (HPC) systems is an established methodology in a wide range of fields not only in traditional computational sciences as physics, chemistry, astrophysics, but also becoming more and more important in biology, economic sciences, and even humanities. The total execution time of an application is rapidly approaching the mean time between failures of large HPC systems. Commonly, only a small part of the system will be affected by the hardware fault, but usually all of the application will crash. Application developers can therefore no longer ignore system faults and need to take fault-tolerance and application resiliency into account as part of the application logic. A necessary step is to store intermediate result as well as the current internal state of the application to allow restarting the application at a later time, which is commonly referred to as checkpointing.

In practice, however, the sheer size of simulation data and the limited I/O bandwidth prohibit dumping checkpoints of the full application state or even full compute node memory at high frequency [2]. Checkpointing frequency is therefore chosen to satisfy requirements of the scientific analysis of the simulation data without any safety net for system failure. It is noted, however, that most computational experiments, i.e., simulations, are by definition sufficiently robust to allow drawing similar or equal scientific conclusions if initial or boundary conditions – and by extension intermediate results – are changed slightly within use-case specific limits. Application-level checkpointing of suitably aggregated intermediate results is therefore being considered as a promising technique to improve the resiliency of scientific applications at relatively low cost of resources. The application developer or end user, purposefully discards most of the intermediate data and checkpoints only those data which are absolutely essential for later reconstruction of a sane simulation state. An example would be to store mean values of given quantities, other suitable higher-order moments of the distribution of the quantities, or leading terms of a suitable expansions. Note however, that the nature of the reconstruction data is fully application and even use-case specific. The application at its restart will use this data to reconstruct the state in the part of the application that was lost to the failure, while keeping the full, precise data in the reset of system which was not affected by the fault.

Previous work in [1] suggests a method for persisting intermediate results and internal application state. The method uses idioms and an interface borrowed from the Message Passing Interface (MPI) [3], which is the most widely used programming model for distributed parallel computing in HPC. This allows users of MPI to integrate our method seamlessly into existing applications at minimal development cost.

This paper is organised into a brief overview of related work in Section II, followed by brief review of our approach to data persistency through MPI semantics in Section III, the demonstration and evaluation of the concepts through a prototypical implementation in Section IV and Section V, respectively, and finally, a short summary of this work in Section VI.

II. Related work

SafetyNet [4] is an example of checkpointing at the hardware level. It keeps multiple, globally consistent checkpoints of the state of a shared memory multiprocessor. This approach has the benefit of lower overhead of runtime but it as additional power and monetary cost. Right now, this approach provides checkpointing solution for a single node only.

In the kernel-level approach, the operating system is responsible for checkpointing, which is done in the kernel space context. It uses internal kernel information to capture the process state and further important information required for a process restart. Berkeley Lab Checkpoint/Restart (BLCR) [5][6] and Checkpoint/Restore In Userspace (CRIU) [7] are two examples of this class. This approach provides a transparent solution for checkpointing but files generated by this approach are large and moving checkpoint files to stable storage takes more time. Another problem associated with this approach is that it requires considerable maintenance and development effort as internals of process state, etc. vary greatly from one OS to another and are prone to change over time.

Checkpointing at the user-level solves the problem of high maintenance effort due to kernel diversity. In this case, checkpointing is done in user-space. All relevant system calls are trapped to track the state of a given process. However, due to the overhead of intercepting system calls it takes more time to complete. Similar to kernel-level, user-level checkpointing needs to save complete process state. So, this approach also suffers from the problem of large file size.

In contrast to the schemes mentioned above, which are transparent to user, application-level checkpointing requires explicit user action. The application developer provides hints to the checkpointing framework. By means of these hints, additional checkpoint code is added to the application. This additional code saves required information and restarts the application in case of failure. Application level checkpointing normally creates smaller size checkpoints as they have knowledge about program state.

One such application-level checkpointing scheme is the library Scalable Checkpoint/Restart (SCR) [8][9]. SCR stores checkpoints temporarily in the memory of neighboring compute nodes before writing them to stable storage. It also includes a kind of scheduler which determines the exact checkpointing time according to system health, resource utilisation and contention, and external triggers. SCR is designed to interoperate with MPI. The application developer uses SCR functions to acquire a special file-like handle. Any data written through this SCR handle is replicated in memory across network neighbours for redundancy and checkpointed to storage transparently in the background at a suitable point in time. The drawback is that application developers have to learn yet another programming interface and add additional, possibly complex, code which is not related to their numerical algorithm. Nonetheless, SCR is a powerful scalable checkpointing tool and thus used in the backend of our prototypical implementation as described in later sections of this work.

Previous extensions to MPI, such as FT-MPI [10] offered the application programmer several possibilities to survive, e.g., leave a hole in the communicator in case of process failure. This particular MPI implementation has been adopted in Open MPI [11]. The Message Passing Interface standard in its current form, i.e., MPI-3 [3], does not provide fault-tolerance. Typically, if a single process of a distributed application fails due to, for instance, catastrophic failure of the given compute node, all other processes involved will eventually fail as well in an unrecoverable manner. Recently, several proposals [12][13][14] have been put forward to mitigate the issue by allowing an application to request notification about process failures and by providing interfaces to repair vital MPI communicators. The application, in principle, can use this interface to return the MPI stack to a sane state and continue operation. However, any data held by the failed process is lost. Notably, this includes any messages that have been in flight at the time of the failure.

III. Persistent MPI communication

In this paper, we present an approach that allows application developers to persist, both, essential locally held data and the content of essential messages between processes. Unlike other models, we use idioms that are familiar to any MPI developer. In fact, we add a single function which returns a MPI communicator with special semantic meaning. Then, the programmer continues to use familiar send and receive MPI calls or collective operations to store data and messages persistently or to retrieve them during failure recovery.

A. Background

In MPI, any process is uniquely identified by its *rank* in a given *communicator*. A communicator can be thought of a ordered set of processes. At initialisation time, MPI creates the default communicator, `MPI_COMM_WORLD`, which includes all processes of the application. New communicators can created as subset of existing ones to allow logically grouping processes as required by the application. Collective MPI operations, as for instance a broadcast or scatter, take a communicator as argument and necessarily require the participation of all the processes of the given communicator. In addition, some collective operation single out one processes which is identified by its rank in the respective communicator. Also, point-to-point communication routines take a communicator as argument. In send operations, the target of a message is passed as rank relative to the given communicator argument. The destination of receive operations is given analogously.

In addition, most MPI communication routines require the specification of the so-called *tag* which allows the programmer to classify different message contents. A tag may be thought of as a P. O. Box or similar. Finally, MPI messages are delivered in the same order they have been issued by the sender. Any MPI message can thus be uniquely identified by the signature tuple `(comm, src, dst, tag)` and a sequence number that orders messages with the same signature. The signature is composed of a communicator, `comm`, the rank of the message source, `src`, the rank of the destination, `dst`, and the message tag `tag`.

B. Persistent communicators and proposed idioms

The basic idea of our approach is very simple. The user marks a communicator as having a special, i.e., persistent, semantics. Any communication issued through a persistent communicator is stored transparently by the MPI library and is available for application restart even after failure (see Figure 1). In contrast to no-persistent communicators, the message is not immediately delivered. An MPI process may thus persist any data and application state by sending it to itself through a persistent communicator. In case of failure the data is simply restored by posting a receive operation on the persistent communicator. Moreover, a process may persist data for any other process by sending a message targeted to the other process through a persistent communicator.

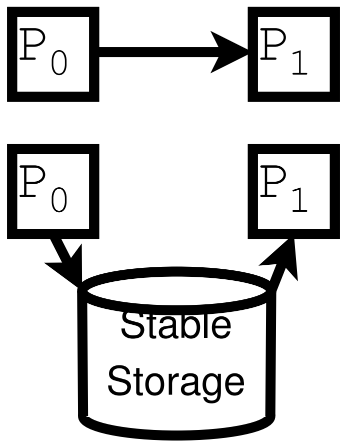

Figure 1. Illustration of communication between two processes, `P0` and `P1`, through regular communicators (top) versus through persisten communicators (bottom).

```
 1  #define VALUE_TAG
 2  const char mykey[] = "Run␣A,␣June␣6␣2015";
 3  MPI_Comm persistent;
 4  int value = 3;
 5
 6  MPI_Comm_persist(MPI_COMM_SELF, info,
 7      mykey, persistent);
 8
 9  if (failure())
10    MPI_Recv(&value, 1, MPI_INT, rank,
11      VALUE_TAG, persistent, &status);
12  else
13    MPI_Send(&value, 1, MPI_INT, rank,
14      VALUE_TAG, persistent);
```

Figure 2. Simple example of data persistency

A communicator is marked as persistent by calling the routine **`MPI_Comm_persist`**, which has the following signature

```
int MPI_Comm_persist(MPI_Comm comm, char *key,
  MPI_Info info, MPI_Comm *persistentcomm)
```

Here, `persistentcomm` is a pointer to memory, which will hold the newly created persistent communicator. It is derived from the existing communicator `comm` and will consist of the same processes, etc. A user-provided string `key` shall uniquely identify this particular application run or instance in case part of it needs to be restarted after a fault occurred. Essentially this serves as a kind of session key. Finally, the info object `info` may hold additional information for the MPI library, as for instance hints where to store data temporarily, or the size of the expected data volume.

A simple example how to persist data is shown in Figure 2. On line 6, the persistent communicator `persistent` is derived from `MPI_COMM_SELF` which is a pre-defined communicator consisting of just the given process. The routine `failure()` shall return `TRUE` if this process is being restarted after a fault. If this is not the case, the application will persistently store the content of the variable `value` by sending a message to itself on line 13. If a fault occurred the application instead will restore the content of the variable `value` by receiving it from itself on line 10.

Our approach also allows to replay or log communication between processes in the case of faults. The programmer simply derives persistent communicators from all relevant communicators and then mirrors every send operation done on a non-persistent communicators with the persistent one. Receive operations are posted on the persistent communicator as necessary by the failed process only. The reduce the amount of additional code one could also allow transparent persistency. In this case a persistent communicator would persist data and also actually deliver data as expected from a non-transparent one. For simplicity, we will not use this facility for the rest of the paper and use persistent communication explicitly.

The core of a somewhat more elaborated example is shown in Figure 3. For the sake of simplicity, we assume that the application is executed with only two processes. This fictitious algorithm evolves for several iterations a very large array of data through a complex calculation `compute` (line 35). At any given point in time, one can however `aggregate` the data into a single value `seed` (line 45). In turn, `seed` can be used to reconstruct the data array with sufficient accuracy by calling `init_data(seed)` (line 25). The algorithm requires to exchange boundary conditions between processes. The first element of the local array is send to the other process, where it replaces the last element (line 38). The system shall provide a function `failure()`, which notifies fault conditions.

The state of the application is given by the iteration counter `i` of the for loop on line 34. This value is persisted by sending a message to oneself (line 50) at the end of each iteration. The algorithm requires the persistence of the `seed`, again by a message to oneself on line 49. Finally, the exchange of boundary conditions is logged on line 42.

In case of failure, the failed process is restarted and restores its internal state (line 22) and the aggregate (line 21) that is used to reconstruct the data array (line 25). The same initialisation operation had been executed by the surviving process with initial values at the original start of the application. The failed process also retrieves the last boundary value received from the other process (line 29). Then it enters the main loop with the correctly restored iteration counter and resumes normal operation in parallel to the surviving process.

IV. Prototypical implementation

In this section, we briefly outline a prototypical implementation of the proposed interface.

A. *Implementation concerns*

Our proposed persistent communicator semantics is relatively easy to implement. As explained in Section III-A, any given MPI message is uniquely identified by its signature and the sequential ordering. In addition, the user has specified a unique session key at the time of creation of the persistent communicator. Together these are used to store any persistent message content in a suitable stable storage. This could be for instance the memory of a neighbor MPI process (or several for redundancy), remote network filesystems, or any data base. After the failure, the application is restarted with the same session key and thus able to map messages to the state before the fault. In our prototype, we persist messages using the SCR library and leave the details to its automatics.

```
#define SIZE VERY_LARGE
#define SESSION 1001

int rank, other;
float data[SIZE], boundary, seed=321;
int iter = 0;
MPI_Comm persistent, world;

MPI_Init();
world = MPI_COMM_WORLD;
MPI_Comm_rank(world, &rank);
if (rank==0)
  other = 1;
else
  other = 0;
MPI_Comm_persist(world, &info, SESSION,
    &persistent)

if (failure()) {
  // retrieve seed and iter
  MPI_Recv(&seed, rank, SEEDTAG, persistent);
  MPI_Recv(&iter, rank, ITERTAG, persistent);
}

init_data(data, seed);

if (failure()) {
  // retrieve boundary conditions
  MPI_Recv(data[SIZE-1], other, BNDTAG,
    persistent);
}


for (int i = iter; i<10, i++) {
  compute(data);

  boundary = data[0];
  MPI_Sendrecv(&boundary, other, BNDTAG,
     data[SIZE-1], other, BNDTAG,
     MPI_COMM_WORLD);
  // store boundary for recovery
  MPI_Send(&boundary, other, BNDTAG,
    persistent);

  seed = aggregate(data);
  printf("%i␣%i␣%f\n", rank, i, seed);

  // store state and aggregate for recovery
  MPI_Send(&seed, rank, SEEDTAG, persistent);
  MPI_Send(&i, rank, ITERTAG, persistent);
}

printf("Final:␣%i␣%f", rank, seed);
MPI_Finalize();
```

Figure 3. A simple MPI program with persistency for 2 processes

```
#include "scr.h"

int MPI_Init(int *argc, char ***argv) {
  int scr_rc;

  // Regular MPI_Init stuff.
  // Assumes success so far.

  scr_rc = SCR_Init();

  if (SCR_SUCCESS != scr_rc) {
    // and error occurred,
    // delegate to OpenMPI for abort.
    return MPI_ERROR_HANDLER();
  }
  return MPI_SUCCESS;
}

int MPI_Finalize() {
  int scr_rc;

  // shutdown SCR first
  scr_rc = SCR_Init();
  if (SCR_SUCCESS != scr_rc) {
    // and error occurred,
    // delegate to OpenMPI for abort
    return MPI_ERROR_HANDLER();
  }

 // Regular MPI_Finalize stuff.
}
```

Figure 4. Illustrative code for initialisation and shutdown of SCR inside the respective MPI methods

Incoming persistent messages with the same signature, and thus different sequence number, shall overwrite the previously stored one. However, one could also implement a stack of user-defined depth and store a history of messages, which are retrieved in order of storage or in reverse. Such schemes could be facilitated by additional parameters provided in the info object at the time of creation of the persistent communicator. For the sake of simplicity of our prototypical implementation, we have chosen the first approach: incoming messages on a persistent communicator overwrite any received previously message with same signature.

We have implemented our prototype on top of OpenMPI v1.10.0. OpenMPI is a very modular implementation of the MPI standard and thus very easy to extend. We have further used the latest version of SCR.

B. Initialisation & shutdown

Users of our persistent message logging shall not have to invoke any method for initialisation or shutdown other than the usual MPI interfaces, i.e., **`MPI_Init`**`()` and **`MPI_Finalize`**`()`, respectively.

However, any application wishing to use SCR needs to invoke the method `SCR_Init()` to initialise the library before invoking any other SCR method. Further, such initialisation of SCR needs to happen after MPI initialisation. Similarly, SCR expects to be shut down by invocation of the method `SCR_Finalize()` before invocation of **`MPI_Finalize`**`()`.

```
// extend communicator object
struct ompi_communicator_t {
  // Regular OpenMPI code.
  int persistFlg;        // >0 if persistent
  char *key;             // session key
  ompi_info_t persistInfo; // arguments
};

int MPI_Comm_persist(MPI_Comm comm,
    char *key,
    MPI_Info info,
    MPI_Comm *newcomm) {

  int rc;

  // duplicate communicator
  rc = MPI_Comm_dup(comm, newcomm);

  // flag as persistent and attach info
  (*(*newcomm)).persistFlg  = 1;
  (*(*newcomm)).key         = key;
  (*(*newcomm)).persistInfo = info;

  if (MPI_SUCCESS != rc) {
    // and error occurred,
    // delegate to OpenMPI for abort
    return MPI_ERROR_HANDLER();
  }
  return MPI_SUCCESS;
}
```

Figure 5. Illustrative code of the method to create a communicator with persistency semantics.

In order to hide this from the user, we have modified the respective MPI methods to take care of SCR initialisation and shutdown as shown schematically in Figure 4. SCR Initialisation is done at the very end of the MPI initialisation method; similarly, SCR shutdown is done right at the beginning of the MPI initialisation method.

An alternative initialisation scheme, is to delay SCR initialisation up to the point where the first communicator is marked for persistency, i.e., the first call to the method **MPI_Comm_persist**(), or even delayed further to the point where the persistent communicator is used for the first time for sending or receiving a message. The advantage of both these approaches is that SCR is only initialised if persistent message logging is actually used in the application. On the other side, the overhead of repeatedly checking if SCR has been initialised already is presumably non-negligible. In any case, SCR needs to be shutdown together with MPI, as there is no other way to infer the last usage of any persistent communicator.

C. Setting up persistent communicators and passing additional arguments

Our proposed scheme is based on using communicators that have been marked by the user as being special. Setting up such a special communicator with persistent semantics is done through the method **MPI_Comm_persist**(). Figure 5 shows a sketch of our implementation of this routine. In order to designate a given communicator as persistent we have extended the definition of OpenMPI's internal communicator data-structure **struct ompi_communicator_t** and added the flag `persistFlg`. We also added a further field `key` to hold the user-specified session key. The meaning of the last additional field `persistInfo` will be explained a little further down this section.

Essentially, setting up a persistent communicator is down by first duplicating the user-provided non-persistent communicator `comm` using the standard MPI functionality **MPI_Comm_dup**(). Then the communicator is marked as persistent by setting the flag `persistFlg` and storing the session key `key` in the communicator object. Finally, if any of the previous steps produced an error, the implementation delegates to OpenMPI's error handler, otherwise it returns successfully.

In addition, the user shall be able to pass additional arguments or hints during setup of persistent communicators. We have decided to follow the same approach for user-hints as in other parts of MPI and exploit the so-called **MPI_Info** objects. Basically, these info objects are a set of user-defined, arbitrary key-values pairs which have semantic significance only in specific context and ignore otherwise. Note that this is also intended as a way to introduce future extensions to our proposal. To that end the method **MPI_Comm_persist**() also takes an argument `info` of type **MPI_Info**. This argument is stored in the corresponding field of the communicator object for later use. In principle, an advanced implementation of our proposal might check the contents of this object at setup time; our prototype just ignores them at this point.

D. Message logging and retrieval

Most of the programme logic required for our scheme sits in the actual communication primitives. As our prototype is intended only as proof-of-concept, we have implemented our proposal only for the two main communication routines **MPI_Send**() and **MPI_Recv**(), as shown schematically in Figure 6. It is trivial to extend the implementation to non-blocking communication primitives or the other communication modes such as buffered and synchronous.

The first thing the communication routines do, is check if the communicator for this messaging request is flagged as persistent. If it is not, processing of the message is delegated to the regular MPI routine. If the communication takes place on a persistent communicator, though, we construct an unambiguous envelope address from the MPI message signature `(comm, src, dest, tag)` (which uniquely identifies a MPI message, see Section III-A), and the user specified session key, which is retrieved from the communicator. The envelope can be something like a string concatenation or a hash function. The next step consists in calculating the total message volume from size of the given MPI datatype `dtype`, which is determined by calling internal MPI services, and the number of such data items `count`. Finally, we pass control to the method `persist()` and `unpersist()`, for **MPI_Send**() and **MPI_Recv**(), respectively, which takes care of actually persisting and retrieving data.

Figure 7 illustrates the implementation of the routines which are used to persist and retrieve a given message buffer through SCR, respectively. In order to persist messages! we register a checkpoint with SCR by calling

```c
#include "scr.h"

int MPI_Send(const void *buf, \
       int count, MPI_Datatype dtype, \
       int dest, int tag, MPI_Comm comm) {

  if (!comm->persistentFlg) {
    // normal send request
    return _MPI_Send(buf, count, \
             dtype, dest, tag, comm);
  }

  // persistent send request from here

  // construct envelope
  src = MPI_Rank(comm);
  key = comm->key;
  env = envelope(comm, src, dest, tag, key);

  // calculate message size
  msize = count * _OMPI_sizeof(dtype);

  // persist buffer
  scr_rc = persist(buf, msize, env);

  if (SCR_SUCCESS != scr_rc) {
    // and error occurred,
    // delegate to OpenMPI for abort.
    return MPI_ERROR_HANDLER();
  }
  return MPI_SUCCESS;
}

int MPI_Recv(void *buf, \
       int count, MPI_Datatype dtype, \
       int src, int tag, MPI_Comm comm, \
       MPI_Status *status) {

  if (!comm->persistentFlg) {
    // normal receive request
    return _MPI_Recv(buf, count, \
             dtype, src, tag, comm, \
             status);
  }

  // persistent recv request from here

  // construct envelope
  dest = MPI_Rank(comm);
  key = comm->key;
  env = envelope(comm, src, dest, tag, key);

  // calculate message size
  msize = count * _OMPI_sizeof(dtype);

  // unpersist buffer
  scr_rc = unpersist(buf, msize, env);

  if (SCR_SUCCESS != scr_rc) {
    // and error occurred,
    // delegate to OpenMPI for abort.
    return MPI_ERROR_HANDLER();
  }
  return MPI_SUCCESS;
}
```

Figure 6. Illustrative code for dealing with persistent communicators inside MPI communication methods.

```c
int persist(void *buf, int msize, \
             char *env) {
  char filename[SCR_MAX_FILENAME];
  FILE *fh;

  // register checkpoint with SCR
  SCR_Start_checkpoint();

  // ask SCR for full filename and open
  SCR_Route_file(env, filename);
  fh = open(filename, "w");

  // hand data over to SCR
  fwrite(buf, 1, msize, fh);

  // disengage from SCR
  fclose(fh);
  SCR_Complete_checkpoint();

  return success();
}

int unpersist(void *buf, int msize, \
               char *env) {
  char filename[SCR_MAX_FILENAME];
  FILE *fh;

  // ask SCR for full filename and open
  SCR_Route_file(env, filename);
  fh = open(filename, "r");

 // retrieve message buffer and disengage
  fread(buf, 1, msize, fh);
  fclose(fh);

  return success();
}
```

Figure 7. Illustrative code to persist communication buffers through SCR.

`SCR_Start_checkpoint()` at the beginning of `persist()`. Next, we ask SCR for a full filename path, which is constructed from the unique envelope string described in the previous paragraph. All (write) operations on this file are routed through SCR and form part of the register checkpoint. We use this facility to store the message buffer. The final call to `SCR_Complete_checkpoint()` commits all data and initiates replication across neighbour nodes as well as storage to disk. The routine for retrieving messages from persistent storage, i.e., `unpersist`,is a bit simpler. SCR is involved only to get the SCR filename as above. The message buffer is than read directly via POSIX file operations without intervention by SCR.

V. Experimental evaluation

In this section, we present a demonstration that our proposed idioms are sufficient to implement user-level fault-tolerance in applications. To that end, we developed a small scientific application, namely heat transfer, and implemented that with Python on top of our MPI prototype. Further, we use this demonstrator to show that the overheads incurred by persisting data through MPI semantics behave as expected.

```
import mpi4py                                       1
from heat import Heat2D                             2
                                                    3
comm = MPI.COMM_WORLD                               4
persist = MPI.Comm_persist(world)                   5
                                                    6
myself = persist.Get_rank()                         7
                                                    8
task = Heat2D()                                     9
                                                    10
while time < end_time:                              11
  # check for failure                               12
  if task.failure():                                13
    persist.Recv(state, myself, STATE_TAG)          14
    task.restore_local_state(state)                 15
                                                    16
  # normal operation                                17
  task.exchange_boundaries(comm)                    18
  task.solver()                                     19
                                                    20
  # save state for fault-tolerance                  21
  if time%interval == 0:                            22
    state = task.get_local_state()                  23
    persist.Send(state, myself, STATE_TAG)          24
                                                    25
  # progress time                                   26
  time += time_step_size                            27
```

Figure 8. Illustration of the python implementation of the heat transfer problem used for evaluation.

A. Experimental setup

We have tested our prototypical implementation on a Cray XC40 supercomputing system running the Cray programming environment CCE 8.4.3. The application code was implemented with Python 2.7.8, and used NumPy 1.9.0 as well as MPI4Py 2.0.0. The prototype was built on top of OpenMPI 1.10.0 and the latest SCR commit 1e8358f from GitHub. The nodes of the Cray XC40 consists of two Intel Haswell E5-2680v3 sockets with 12 cores each. For the experiments we ran OpenMPI over the TCP conduit, as stock OpenMPI does not support the native Cray interconnect.

The application code solves the well known heat diffusion equation. The schematic code in Figure 8 illustrates the parts relevant to this paper only. Most of the programmes business logic is encapsulated in the class `Heat2D` and instantiated as object `task`. The application uses two distinct communicators: the regular communicator `comm`, and a persistent communicator `persist` which is derived from `comm` on line 5. To complete the initialisation, each MPI process stores its own rank in the variable `myself`. The main part of the application consists of the time integration loop starting at line 11. In its original, i.e., non-persistent, version the time loop would consist only of exchanging boundary conditions with neighbour process through the communicator `comm` and the actual solver step on lines 18–19, as well as increment of time on line 27.

To achieve fault-tolerance, the local state is determined and stored periodically through the persistent communicator as illustrated on lines 21–24. Note that the definition of a local state, which is suitable for application restart is completely

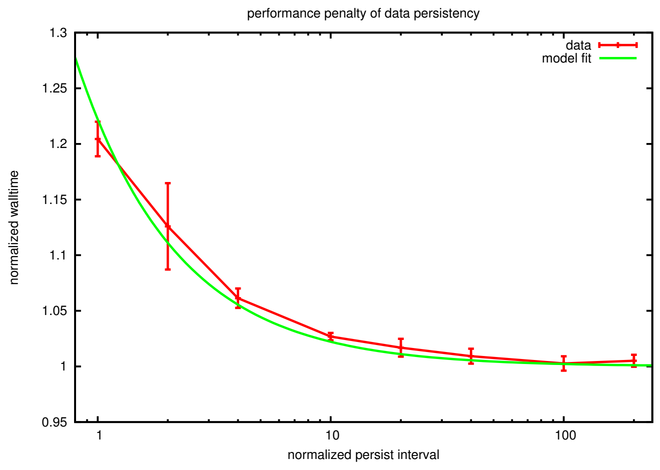

Figure 9. Overhead of our implementation as a function of length of interval between two data persist events.

up to the use case. Here, we have take a straightforward average over the temperatur on the grid. Saving the state is accomplished simply by sending a message to `myself` on the persistent communicator. The user may choose to do this for every times step or at intervals specified through the variable `interval`. In case of failure, the application retrieves the state by receiving a message from `myself` on the persistent communicator. This message is used to restore the local state as shown on lines 12–15.

Clearly, achieving fault-tolerance incurs a runtime overhead for the application. Additional time is spent, in particular, determining if the application state needs saving, aggregating the application state, and the actual cost of persisting it. Part of these overheads are beyond the responsibility of our implementation. However, for the sake on simplicity, we have benchmarked the fault-tolerant code against a version where lines 5, 14–15, and 23–24 were commented out, essentially. So, the measured overhead includes the calculation of the local state, which however should be small.

The overhead should depend on the frequency of persisting the local state, and on the duration of doing so. In our experiment, we have varied the persistency interval, i.e., the variable `interval` in the code above. As we cannot directly control the duration of the persistency operation, we have varied the time taken to calculate a single iteration of the time loop by increasing the problem size. We have expressed the problem size in such a way, that the execution time for a single iteration depends linearly on it. All benchmark experiments have been repeated at least 10 times. The values reported correspond to the average over all runs. The error bars are calculated from the sample variance; error propagation calculus is used for all values calculated from the basic measurements.

B. Results and discussion

In order to study the overheads of our implementation, we have varied the interval at which the local state is persisted. Figure 9 shows the ratio of execution time of the code with persist logic over the original non-fault-tolerant version as a function of the interval. Note that larger interval values correspond to less frequently saved states. As expected, the

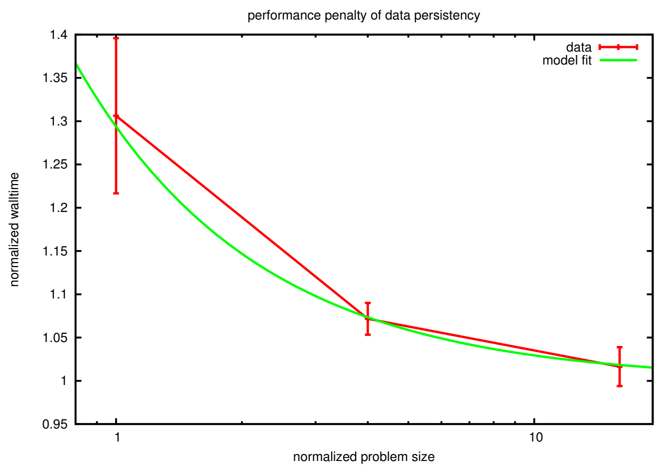

Figure 10. Overhead of our implementation as a function of the application's problem size.

overhead is largest for small intervals and decreases with increasing interval length. We have fitted the experimental data to the expected model curve $T(i) = (i + c_i)/i$. Here, T is the normalised execution time, i the interval length, and c_i a fit parameter. The experimental data is described very well by model; the standard error on the fit parameter c_i is less than 5%. This shows that for large values of the interval the overhead becomes negligible and vanishes asymptotically.

In a second series of experiments, we fixed the interval to unity and varied the problem size as shown in Figure 10. At small problem sizes, the overhead is large, as persisting data takes more time compared to the calculation of an iteration of the algorithm. With increasing problem size, the time taken to persist the application state decreases in relation to the time spent in calculations and the overhead decreases. Again, this can be modelled with a function of the form $T(s) = (s + c_s)/s$. As shown in the figure, the model describes the data very accurately. The error on the fit parameter c_s is less than 3%. From this model, we can again expect that the overhead becomes negligible at sufficiently large problem size.

VI. Conclusions

In this paper, we have presented work in progress on the a method to allow persisting of application data and internal state for fault recovery. Unlike other methods, our approach uses well known MPI semantics. The only addition to MPI is a routine that allows to mark a communicator as persistent. All messages to such a communicator are stored on a stable storage for later usage during failure recovery. We have shown basic idioms of storing and retrieving not only application data, but also internal state of the application and to use message logging to recover messages that have been exchanged with other MPI processes just prior to the fault. To verify that the proposed idioms are sufficient to realise user-level data persistency for fault-tolerance, we have done a prototypical implementation of our interface and demonstrated the concept with a typical scientific application. Finally, we have shown that the overheads of prototypical become negligible for sufficiently large problem sizes or sufficiently large data persistency intervals.

Acknowledgment

This work was supported by the German Research Foundation (DFG) through the German Priority Programme 1648 Software for Exascale Computing (SPPEXA) and the EC H2020 programme through the project MIKELANGELO under Grant Agreement no. 645402.

References

[1] J. Gracia, M. W. Sethi, N. Struckmann, and R. Keller, "Towards data persistency for fault-tolerance using MPI semantics," in Proceedings of International Conference on Advanced Communications and Computation (INFOCOMP 2015). IARIA, 2015, pp. 26–29.

[2] Y. Ling, J. Mi, and X. Lin, "A variational calculus approach to optimal checkpoint placement," IEEE Trans. Computers, vol. 50, no. 7, 2001, pp. 699–708.

[3] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," September 21st 2012, available at: http://www.mpi-forum.org [retrieved: May, 2016].

[4] D. J. Sorin, M. M. K. Martin, M. D. Hill, and D. A. Wood, "Safetynet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery," SIGARCH Comput. Archit. News, vol. 30, no. 2, May 2002, pp. 123–134.

[5] J. Duell, P. Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," Future Technologies Group, white paper, 2003.

[6] J. Cornwell and A. Kongmunvattana, "Efficient system-level remote checkpointing technique for blcr," in Proceedings of the 2011 Eighth International Conference on Information Technology: New Generations, ser. ITNG '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1002–1007.

[7] CRIU project, "Checkpoint/Restore In Userspace – CRIU," 2015, available at: http://http://www.criu.org/Main_Page/ [retrieved: May, 2016].

[8] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, ser. SC '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–11.

[9] K. Mohror, A. Moody, and B. R. de Supinski, "Asynchronous checkpoint migration with mrnet in the scalable checkpoint / restart library," in IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN 2012, Boston, MA, USA, June 25-28, 2012. IEEE, 2012, pp. 1–6.

[10] G. E. Fagg et al., "Fault tolerant communication library and applications for high performance," in Los Alamos Computer Science Institute Symposium, Santa Fe, NM, Oct. 2003, pp. 27–29.

[11] E. Gabriel et al., "Open MPI: Goals, concept, and design of a next generation MPI implementation," in Proceedings of the 11^{th} European PVM/MPI Users' Group Meeting, ser. LNCS, D. Kranzlmüller, P. Kacsuk, and J. Dongarra, Eds., vol. 3241. Budapest, Hungary: Springer, Sep. 2004, pp. 97–104.

[12] W. Bland, A. Bouteiller, T. Hérault, J. Hursey, G. Bosilca, and J. J. Dongarra, "An evaluation of user-level failure mitigation support in MPI," in Recent Advances in the Message Passing Interface - 19th European MPI Users' Group Meeting, EuroMPI 2012, Vienna, Austria, September 23-26, 2012. Proceedings, 2012, pp. 193–203.

[13] W. Bland, A. Bouteiller, T. Hérault, G. Bosilca, and J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," IJHPCA, vol. 27, no. 3, 2013, pp. 244–254.

[14] J. Hursey, R. Graham, G. Bronevetsky, D. Buntinas, H. Pritchard, and D. Solt, "Run-through stabilization: An mpi proposal for process fault tolerance," in Recent Advances in the Message Passing Interface, ser. Lecture Notes in Computer Science, Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, Eds. Springer Berlin Heidelberg, 2011, vol. 6960, pp. 329–332.