Towards Data Persistency for Fault-tolerance Using MPI Semantics

José Gracia,* M. Wahaj Sethi,†* Nico Struckmann,* Rainer Keller‡
*High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Germany
†e.solutions GmbH, Ingolstadt, Germany
‡University of Applied Sciences, HfT Stuttgart, Germany

***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.**

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

I. INTRODUCTION

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 all intermediate results at high frequency [1]. Checkpoints are therefore chosen to satisfy requirements of the scientific analysis of the simulation data. However, 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 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.

In this paper we present a method for persisting intermediate results and internal application state. Our proposed method uses idioms and an interface borrowed from the Message Passing Interface (MPI) [2], 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 organized into a brief overview of related work in Section II, followed by a our approach to data persistency through MPI semantics in Section III, and finally a short summary of this work in Section IV

II. RELATED WORK

SafetyNet [3] 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) [4][5] and Checkpoint/Restore In Userspace (CRIU) [6] 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

 ISBN: 978-1-61208-416-9

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) [7][8]. 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 utilization and contention, and external triggers. SCR is designed to interoperate with MPI. The application developer uses SCR functions to mark important data which is then checkpointed 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.

Previous extensions to MPI, such as FT-MPI [9] 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 [10]. The Message Passing Interface standard in its current form, i.e., MPI-3 [2], 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 [11][12][13] 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.

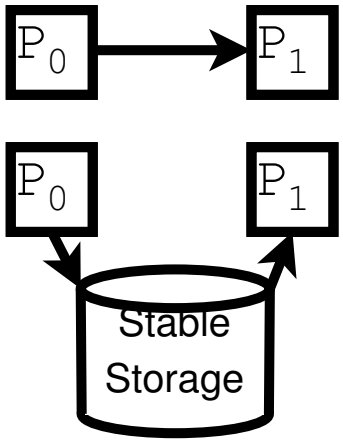

Figure 1. Illustration of communication between two processes, `P0` and `P1`, through regular communicators (top) versus through persisten communicators (bottom).

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 initialization 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

```
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

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.

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

```
#define SIZE VERY_LARGE                                  1
#define SESSION 1001                                     2
                                                         3
int rank, other;                                         4
float data[SIZE], boundary, seed=321;                    5
int iter = 0;                                            6
MPI_comm persistent, world;                              7
                                                         8
MPI_Init();                                              9
world = MPI_COMM_WORLD;                                  10
MPI_Comm_rank(world, &rank);                             11
if (rank==0)                                             12
  other = 1;                                             13
else                                                     14
  other = 0;                                             15
MPI_Comm_persist(world, &info, SESSION,                  16
    &persistent)                                         17
                                                         18
if (failure()) {                                         19
  // retrieve seed and iter                              20
  MPI_Recv(&seed, rank, SEEDTAG, persistent);            21
  MPI_Recv(&iter, rank, ITERTAG, persistent);            22
}                                                        23
                                                         24
init_data(data, seed);                                   25
                                                         26
if (failure()) {                                         27
  // retrieve boundary conditions                        28
  MPI_Recv(data[SIZE-1], other, BNDTAG,                  29
    persistent);                                         30
}                                                        31
                                                         32
                                                         33
for (int i = iter; i<10, i++) {                          34
  compute(data);                                         35
                                                         36
  boundary = data[0];                                    37
  MPI_Sendrecv(&boundary, other, BNDTAG,                 38
     data[SIZE-1], other, BNDTAG,                        39
     MPI_COMM_WORLD);                                    40
  // store boundary for recovery                         41
  MPI_Send(&boundary, other, BNDTAG,                     42
    persistent);                                         43
                                                         44
  seed = aggregate(data);                                45
  printf("%i␣%i␣%f\n", rank, i, seed);                   46
                                                         47
  // store state and aggregate for recovery              48
  MPI_Send(&seed, rank, SEEDTAG, persistent);            49
  MPI_Send(&i, rank, ITERTAG, persistent);               50
}                                                        51
                                                         52
printf("Final:␣%i␣%f", rank, seed);                      53
MPI_Finalize();                                          54
```

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

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

 ISBN: 978-1-61208-416-9

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 require 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 fore 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), the aggregate (line 21) which is used to reconstruct the data array (line 25). The same initialization 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). The it enters the main loop with the correctly restored iteration counter and resumes normal operation in parallel to the surviving process.

C. Implementation concerns

Our proposed persistent communicator semantics is relatively easy to implement. As explained in 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 one (or several for redundancy) neighbor MPI processes, remote network filesystems, or any data base. In fact, one could persist the data using the SCR library and leave the details to is automatics. After the failure, the application is restarter with the same session key and thus allow to map messages to the state before the fault.

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.

IV. 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.

Acknowledgment

This work was supported by the German Federal Ministry of Education and Research (BMBF) through the project FEToL (Grant Number 01IH11011F).

References

[1] 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.

[2] MPI Forum, "MPI: A Message-Passing Interface Standard. Version 3.0," September 21st 2012, available at: http://www.mpi-forum.org [retrieved: May, 2015].

[3] 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.

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

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

 ISBN: 978-1-61208-416-9