DMTCP Checkpoint/Restart of MPI Programs via Proxies

Gregory Price
March 28, 2018

Abstract

MPI accomplishes portable, standardized message-passing between processes by exposing a standard API that hides the implementation of the underlying mechanism for message passing. Until now, checkpointing an MPI program required knowledge of these underlying mechanisms. Through the addition of a proxy, we demonstrate that MPI programs can be checkpointed and restarted regardless of the MPI implementation utilized. Further, proxies may enable MPI programs to be checkpointed on one MPI implementation, and restarted on another.

1 Introduction

This work has largely been produced using examples from other DMTCP plugins from the core dmtcp library [1]. It was implemented and tested against MPICH [2], with examples from mpi-tutorial.com [3].

MPI provides the HPC community a standardized way to pass messages between processes in parallel computation. Each MPI implementation provides an opaque, standardized interface to send and receive messages; however, each implementation may differ in system requirements and mechanisms used to actually pass the message (i.e., sockets, pipes, shared memory, etc.).

This provides a challenge when checkpoint/restart is desired. Classically, checkpoint/restart is achieved via virtualization of various resources (such as file descriptors, sockets, shared memory, etc.), which are used transparently by the user application. Checkpoint/restart software must then provide support for each potential implementation of MPI; agnosticism is a pipe dream in this reality.

Through the use of MPI proxies and a DMTCP Plugin, we present a solution that provides simple, portable, transparent, and agnostic Checkpoint/Restart. Our implementation requires no changes to application code, and can be deployed on any architecture machine that supports DMTCP.

Our strategy makes one, key, novel observation about how to checkpoint MPI: don’t. By interposing on the MPI library, and moving all MPI procedures into a separate process, we only need to checkpoint a single, ephemeral interface between the application program and the proxy. Then, we simply, transparently translate MPI calls into calls to our MPI proxy.

Restart presents a separate problem, one which is fairly understood within the realm of checkpoint/restart: How do we handle messages in flight? There are two options for managing the situation.

1. Log and replay of messages on restart
2. Drain the network of in-flight messages, and cache them for restart

Logging many messages requires additional (potentially significant) overhead throughout the lifetime of the computation, while draining the network only incurs a cost at the time of checkpoint. We employ network draining as our mechanism of choice, under the assumption that a one-time cost during checkpoint is more tolerable to HPC professionals. This overhead can also be easily controlled through changing how often a checkpoint is created.

Finally, we believe solving this checkpoint/restart issue for MPI via the combination of library interposition (via DMTCP plugins) and a proxy provides a compelling case for using this strategy elsewhere. Other technologies that provide a standardized interface to an external service can also benefit from checkpoint/restart via the use of proxies.
2 Background

MPI provides two main services:

1. Creation of processes (called ranks) for distributed computation
2. Message passing between these ranks

A coordinator is used for exchanging information about rank identification, how to build direct communication paths between ranks, and what groups have been created for publish-subscribe style communications (Broadcasting to a subset of ranks). It is useful to visualize this relationship in Figure 1, as one possible implementation of an MPI coordinator.

![Figure 1: The communication structure of a simple MPI application](image1)

From here, we can see that there is a parent-child process relationship. The Coordinator is a parent to both Rank 0 and Rank 1. Also, there are at least three “communication interfaces” created between these three programs: Interfaces between each rank and the coordinator, and between each rank.

Each rank has a separate “MPI Library”, which is used by the application program to interact with the coordinator and other ranks. We call this an *Active MPI Library*. An “Active” library is one whose functions are loaded and executed, whereas a “passive” library is one whose functions may be referenced but never used.

![Figure 2: MPI library in User Application](image2)

Prior work on checkpointing MPI programs had been built on the assumption that all external resources within the entire system must be virtualized. We can visualize this with a “checkpoint boundary” for an arbitrary MPI computation.

Each relationship and communication mechanism within the checkpoint boundary will increase the complexity of a successful checkpoint/restart approach as the number of potential communication mechanisms used by MPI implementations increases. (The “checkpoint boundary” encapsulates all the resources that must be recorded and/or restored upon restart.)
3 Our Approach

We seek to avoid checkpointing any implementation-specific MPI code, and instead create a network with a checkpoint boundary that looks as follows:

![Checkpoint Boundary Diagram](image_url)

Figure 4: Our Checkpoint Boundary

This reduces the total number of interfaces needed to be virtualized, and completely segregates the MPI implementation from the user application. The MPI library used by the user application is no longer “active”, and is replaced by a DMTCP plugin that provides the virtual interface to the “active” library made available by the proxy.

4 challenges

By introducing an external proxy, a one-step message-passing mechanism is broken into two steps. This introduces a temporal issue that was not present in other checkpointing solutions. We must deal with two challenges that must be solved in order to successfully checkpoint and restart:

1. In-flight Data. Rank 0 sends Rank 1 a message, the message has been received by Proxy 0, but has not been transferred to Proxy 1 or Rank 1. A checkpoint is initiated, leaving the message stuck “in flight”.

---

**Note:** The diagram labeled “Checkpoint Boundary” is included in the text but is not explicitly mentioned above. It is a visual representation of the checkpoint boundary discussed in the text. The figure is crucial for understanding the implementation of the approach described.
2. Proxy State. Rank 0 makes a call to receive a message, but Rank 1 has not sent the message yet. Rank 0 must re-issue the call in order to receive on restart.

In both cases, we recognize that a potential for failure occurs when the proxy has a “state” that is not being recorded due to our chosen checkpoint boundary. We revert to the two basic techniques discussed previously (Drain, and Log-and-Replay) to solve these in the simplest way possible.

For “in-flight data”, we drain the network of all in-flight messages, and cache this information in the receiving processes’ memory space. In order to know when all in-flight data has been successfully drained, we utilize the DMTCP coordinator to share the number of messages that each rank has sent and received. When draining, we continue to increment these numbers. When the numbers are equivalent, we know that there is no longer any data in flight, and thus we can move forward with checkpointing. This heuristic was previously used in the PhD thesis of Jiajun Cao for checkpointing over InfiniBand [5].

For Proxy-State replication, we break messages into two basic categories:

1. Administrative messages
2. Message actions

Administrative messages are messages between the rank and the MPI coordinator to either retrieve information about the current configuration of the MPI ranks, or to create new configurations (such as new groups of ranks). Configuration messages must be replayed to ensure that the “Active” MPI library in the proxy is in the same state as when we initiated the checkpoint.

Message actions are calls such as MPI_IProbe and MPI_Recv, where the user application is trying to retrieve either data, or information about the state of the network. For example, MPI_IProbe is used to check if there is a message waiting to be received. The relevant information for these message actions must be cached at checkpoint time. The simplest example is the drain mechanism for in-flight data — these messages are cached for use on restart. A call to MPI_Recv must check the cache for a buffered message before checking the proxy for a buffered message.

5 API Support

Currently only the following calls to the MPI interface are fully supported.

- MPI_Init
- MPI_Finalize
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Type_size
- MPI_Send
- MPI_Recv
- MPI_Probe
- MPI_Iprobe
- MPI_Get_count

Future work will expand support to the following API calls:

- Non-blocking Asynchronous Send / Receive
  - MPI_Isend
  - MPI_Irecv
  - MPI_Test
- Collective Communication
- MPI_Bcast
- MPI_Barrier
- MPI_Scatter
- MPI_Gather
- MPI_Allgather
- MPI_Reduce
- MPI_Allreduce

- Communicators and Groups
  - MPI_Comm_group
  - MPI_Group_incl
  - MPI_Comm_create_group
  - MPI_Group_free
  - MPI_Comm_free

6 Current State

Presently we support checkpointing of simple Send/Receive programs that use any of the supported API calls. We do not currently support asynchronous send and receive (i.e., non-blocking sends and receives). Trivially, we support non-blocking receives by supporting MPI_Iprobe, which allows user applications to determine if a message is waiting to be received. However, support of MPI_Isend will require the caching of additional data (not yet implemented).

With these basic building blocks supported, we can confidently say that the remaining API calls can also be supported — it is now a simple matter of plumbing, and no longer a question of feasibility.

7 Future Work

Along with adding full API support, we also intend to explore the idea of checkpointing an application on one MPI implementation, and then restarting the application on a different MPI implementation. This will be accomplished via two core mechanisms. First, we will provide a virtualized interface to MPI specific resources, such as an MPI_Request id or MPI_Status structure. Second, we will provide a new layer to handle translations between the Passive MPI Library in the MPI Plugin and the Active MPI Library in the MPI Proxy. We believe this approach will make it possible to, for example, checkpoint a program built and run on MPICH, and restart the same program on an OMPi implementation without recompilation.

8 Acknowledgements

Gene Cooperman and Rohan Garg (Northeastern University) provided guidance on DMTCP plugin creation, and in the design of the proxy architecture. The initial idea for the proxy architecture was developed by Rohan Garg, Gene Cooperman and Jiajun Cao [5] during the development of other checkpoint/restart technologies. This report and project reinforces their claim that a proxy architecture for checkpointing can be applicable much more generally.

The members of the High Performance Computing Lab at Northeastern University provided feedback and support during the development of this project.

References

[1] DMTCP. http://github.com/dmtcp/dmtcp.
[2] MPICH. http://www.mpich.org.
[3] MPI Tutorials. http://mpitutorial.com
[4] Douglas Thain, Miron Livny: "Multiple Bypass: Interposition Agents for Distributed Computing, Cluster Computing" 4(1): 39-47 (2001)

[5] Jiajun Cao, Transparent Checkpointing over RDMA-based Networks, PhD thesis, 2017