SAND2020-10900C

Integrating Inter-Node Communication with a Resilient Asynchronous Many-Task Runtime System

Sri Raj Paul
Georgia Institute of Technology
Atlanta, USA
sriraj@gatech.edu

Akihiro Hayashi
Georgia Institute of Technology
Atlanta, USA
ahayashi@gatech.edu

Matthew Whitlock
Georgia Institute of Technology
Atlanta, USA
mwhitlock9@gatech.edu

Seonmyeong Bak
Georgia Institute of Technology
Atlanta, USA
sbak5@gatech.edu

Keita Teranishi
Sandia National Laboratories
Livermore, USA
knteran@sandia.gov

Jackson Mayo
Sandia National Laboratories
Livermore, USA
jmayo@sandia.gov

Max Grossman
Georgia Institute of Technology
Atlanta, USA
max.grossman@gatech.edu

Vivek Sarkar
Georgia Institute of Technology
Atlanta, USA
vsarkar@gatech.edu

***Abstract*—Achieving fault tolerance is one of the significant challenges of exascale computing due to projected increases in soft/transient failures. While past work on software-based resilience techniques typically focused on traditional bulk-synchronous parallel programming models, we believe that Asynchronous Many-Task (AMT) programming models are better suited to enabling resiliency since they provide explicit abstractions of data and tasks which contribute to increased asynchrony and latency tolerance. In this paper, we extend our past work on enabling application-level resilience in single node AMT programs by integrating the capability to perform asynchronous MPI communication, thereby enabling resiliency across multiple nodes. We also enable resilience against fail-stop errors where our runtime will manage all re-execution of tasks and communication without user intervention. Our results show that we are able to add communication operations to resilient programs with low overhead, by offloading communication to dedicated communication workers and also recover from fail-stop errors transparently, thereby enhancing productivity.**

***Index Terms*—Resilience, AMT Runtimes, Habanero C/C++, MPI communication, Fenix, MPI-ULFM**

I. Introduction

Fault Tolerance is one of the significant challenges of exascale computing due to projected increases in soft/transient failures. According to [1], the application failure probability on the Blue Waters system increases by a factor of 20× when the number of nodes is only doubled. This emphasizes the importance of providing failure mitigation mechanisms. One of the insightful conclusions from [1] is that application-level resilience plays an essential role in improving application resiliency.

The most popular application-level resilience technique today is coordinated checkpoint and restart (C/R) typically with bulk-synchronous parallel programming models [2]–[4], where processing elements (PE) cooperatively maintain a consistent global application state. While global recovery models are better suited for hard failures such as node failures, unfortunately they are inefficient for transient local failures because global reactions to locally-maskable failures can incur significant performance overheads. Since the majority of application failures are attributed to local node/process failure [2], local recovery approaches such as Containment Domains (CDs) [5] are essential.

Additionally, given that the complexity of node architectures is increasing, it is important to enable resiliency in emerging parallel programming models such as asynchronous many-task (AMT) programming models [6]–[13]. One of the interesting properties of AMT models is that they decompose an application program into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing the application at the process level (MPI ranks). Such properties enable both a decomposition of work on to heterogeneous node resources as well as more efficient local error recovery, better than bulk-synchronous models. That is, given that tasks represent a small piece of program execution and assuming that failures are manifested as failed or lost tasks, failures can typically be remediated using lightweight mechanisms such as task replay.

In our previous work [14], we introduced a comprehensive approach to enabling application-level resilience in Asynchronous Many-Task (AMT) programming models with a focus on remedying Silent Data Corruption (SDC) that can often go undetected by the hardware and OS. Mainly, this approach offers different resilience techniques including task replay, task replication, algorithm-based fault tolerance (ABFT), and checkpointing in a single parallel programming model: the

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Habanero C/C++ library (HClib).

Since one of the key limitations of the previous work is that it only provides single-node resilience, in this paper, we integrate our resilient AMT programming model with asynchronous MPI communication to enable resiliency across multiple nodes including fail-stop and fail-continue errors. Essentially, our programming model introduces a high-level asynchronous MPI communication API that can be transparently used in different resilience scenarios. This design makes it possible for the application programmer to declaratively express resilience attributes and inter-node asynchronous communication routines with minimal code changes, and to delegate the complexity of efficiently supporting resilience to the HClib runtime system.

To the best of our knowledge, this is the first work to provide the design, implementation, and evaluation of a unified programming model that transparently supports multiple resilience techniques across multiple nodes.

This paper makes the following contributions:

1) Programming model extensions to enable resiliency across multiple nodes by introducing asynchronous communication APIs, which is built on top of an on-node resilient AMT runtime [14].
2) Unified execution of resilient, non-resilient, and asynchronous communication tasks in a single framework.
3) Implementation of our approach as extensions to the Habanero-C/C++ library for many-task parallelism.
4) Performance evaluation of our implementation with synthetic error rates, and analysis of the results.

II. Background

In this section, we discuss the background of resilient programming models and runtimes for HPC.

A. Types of Errors: Fail-stop vs Fail-continue

On HPC systems, failures in application programs are manifested as job failures or incorrect application outputs, resulting in wastage of computing cycles or use of wrong results. From the resilience perspective, the errors that cause these failures are classified into two major categories [15]: fail-stop and fail-continue errors.

Fail-stop errors causes the process to stop or crash, resulting in loss of all computation and data. Typically, failures from fail-stop errors can be detected by the operating system, middleware, or transport layer of HPC systems. For example, in parallel programs written with MPI, a crash of single rank (process) is detected by MPI message passing calls to the lost rank or the underlying process manager interface.

Fail-continue errors causes the process to fail but continue to execute, often due to a transient error. In some cases, the process might decide to continue execution even after detection of errors (Detected Uncorrected Errors - DUE). In other cases, the error goes undetected by the hardware, operating system, and other middleware components, which is referred to as Silent Data Corruptions (SDC). If SDC happens in the lowest bits of a mantissa or in application data that is never referenced after the error, the final output of the program is likely to be unchanged. If the corruption occurs in more significant bits, however, it can impact the final output of the application in a way that is either obvious or subtle. Therefore we need suitable mechanisms to check the correctness of results where precision is critical.

B. Types of Recovery

Next, we discuss the failure recovery techniques targeted at today's dominant SPMD model. In particular, resilience and fault-tolerance for the MPI programming model is discussed.

1) Global Recovery: Today, Coordinated Checkpoint/Restart (C/R) is widely used for global recovery and a few production-ready software packages [2]–[4] are available in the public domain. In a parallel program execution, Coordinated C/R synchronizes all running processes to create a consistent global snapshot of the program, called a checkpoint, which is then stored in persistent storage such as a globally-accessible, networked file system. When failure is detected in the program execution, rollback is initiated. Global recovery can lead to the disproportionate use of computational resources to handle the most common failures occurring on a single thread, process, or node.

A proposed resiliency addition to the MPI specification, User Level Fault Mitigation (ULFM) [16], aims to reduce the overhead of failures by allowing MPI to detect failed ranks and users to modify communicators to exclude those ranks. This enables recovery of a failed rank without doing full process shutdown/restart and instead only doing an in-process recovery and restart. MPI-ULFM is highly performant and versatile, but applications may need significant modifications to implement error handling. Fenix [17], [18] is a framework which leverages MPI-ULFM to provide user-level recovery of failed ranks with a streamlined API that limits code modification to a few locations. Fenix provides two separate interfaces for process and data recovery, and is designed in principle to function with a variety of backends for either with minimal changes to user code when switching.

2) Local Recovery: Along with the emergence of fault-tolerance proposals in the Message Passing Interface (MPI) standard, there has been an emerging idea of the local recovery of parallel programs to overcome the shortcomings of coordinated C/R. This idea is based on the observation that the majority of application failures are attributed to local node/process failure as reported by [2], and that the recovery need only be applied to the corrupted processes without global coordination.

For example, uncoordinated Checkpoint Restart (UC/R) [19] exploits message contents (message logging) exchanged between MPI ranks to enable localized recovery. Another example of local recovery is Containment Domains (CDs) [5] that enables efficient and transparent recovery of applications by providing good abstractions for failure detection and correction.

C. Asynchronous Many-Task Programming and Execution Model

The asynchronous many-task (AMT) model [6]–[13] is a category of programming and execution models proposed as an alternative to the dominant SPMD programming models. AMT programming models and runtime softwares have several common functionalities across different implementations and packages. Typically, these frameworks decompose an application program into small, transferable units of work (many tasks) with associated inputs (dependencies or data blocks) rather than simply decomposing at the process level (MPI ranks). To enable more sophisticated decomposition of a program, the architecture of a typical AMT runtime involves several software components as listed below.

- Tasks
- Data blocks
- Runtime scheduler (task queues, dependency graph and task/data tables)
- Workers (thread/processes)

Despite minor differences between individual AMT implementations, an AMT runtime provides APIs to instantiate these components. The most important features are task and data objects encapsulated with their meta-data representations so that the runtime scheduler can orchestrate these objects. The runtime scheduler consists of task queues and a special construct to represent the task dependencies and monitor the status of task and data objects. Task dependencies can be expressed either explicitly or implicitly. For example, ParSEC [9] employs a static parametric task-graph to express all task dependencies, and the open community runtime (OCR) [13] employs `event` objects to notify of state changes of individual tasks and data objects. OpenMP has supported task parallel computing since version 3.0, and extends the capability in later versions. The latest version of Kokkos [20] supports task parallel computing to extend its performance-portable, data-parallel computing interface. Like Kokkos, HClib [21], described in Section IV, exploits modern C++ features to instantiate tasks and data objects.

The term many-task encompasses the idea that the application is decomposed into many transferable or migratable units of data/work, to enable the overlap of communication and computation as well as asynchronous load balancing strategies. The transferable units and load balancing can be used as a mechanism to support easy incorporation of fault tolerance.

In this paper, we discuss the design of a unified interface for AMT programming models that enables the recovery of distributed applications from Silent Data Corruptions (SDC) as well as fail-stop errors.

III. Design

In this paper, we extend the resilience mechanisms incorporated in the Habanero C++ library (HClib) [21] that was published at Euro-par 2019 [14] to include inter-node communication.

A. Execution Model

Figure 1 shows the high level structure of the execution model for the j_{th} rank of a distributed job, shown as *process[j]*, with *memory[j]* representing that process's locally accessible memory. This rank with its n cores is split between computation and communication worker threads. A few of the cores get allocated for computation workers (shown as executing on *core[1:i]*) and the rest of the cores allocated for communication workers (shown as executing on *core[i+1:n]*).

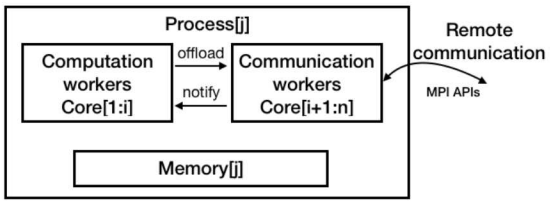

Fig. 1. The internal structure of workers within a single rank.

To enable asynchronous communication, we offload the remote accesses on to a communication worker by spawning a communication task. Then, the communication worker picks up that task and performs the actual remote communication. Once the communication is complete, the communication worker notifies the arrival of data to the computation workers. Since we use separate workers for computation and communication, they can progress without starving the other.

B. Resilience Module Recap

A brief recap of the resilience mechanisms incorporated [14] in HClib is discussed here. Enabling resilience efficiently requires identification of two key points, 1) where in the program and 2) what data we need to check to correct faults. Task boundary is an ideal program location where we can perform error checking and recovery without worrying about the task's internal state or the application's global state. The next step is to identify the data that needs to be verified. For that, we look at data that is going to be used past the task boundary, i.e., the task outputs. Primarily we look at data that is live with respect to task boundary. Since task-based runtimes encourage the use of built-in constructs for task inputs and outputs (for example, Logical regions in Legion [8] and Datablocks in Open-Community-Runtime(OCR) [13]), we can find out the live data at task boundary using the construct used to express task outputs. Similarly C++11 introduced `promise` and `future` constructs to facilitate data transfer across tasks along with synchronization to avoid data races. A `promise` is a thread-safe container with single-assignment semantics to fill its value. The value written to a `promise` is read using its read-only handle called as `future`. Thus `promise` and `future` enable point-to-point synchronization between one source task to many sink tasks.

1) Resilience API Specification: Now we can see how we extended different AMT components, mainly tasks and promise/futures to enable various resiliency techniques. For demonstration purposes, we use `async` as a generic AMT

construct that creates an asynchronous task with a user-provided lambda expression, and `async_await` is a variant of `async` that can wait on a `future`.

Table I shows the incremental extensions that was added to HClib to enable various resilience mechanisms. The additional parameters are shown in red color.

The async_await API is used to create a data-driven task with future_deps as its dependencies. async_await_check is its corresponding resilient API which uses various resilience mechanisms such as Replay, Replication, and ABFT. Further, checkpointing can be added to any of these by separately enabling checkpoint storage and thus keeping the respective API of replication/replay/ABFT effectively unchanged. Please refer [14] for more details.

C. Communication Module

We focus our study on distributed resiliency on MPI as MPI is the most commonly used communication library in HPC. Additionally, the MPI standards committee is working towards enabling resilience for MPI communication in exascale systems. This is very helpful since we can now work on enabling resilience by treating communication as a reliable black box.

This work focuses on the non-blocking MPI send and receive APIs. We currently do not support other APIs such as collectives and so on. We focus on non-blocking communication as it is synergistic with the AMT model since both are capable of launching the operation asynchronously thereby allowing the execution to proceed without waiting for the operation to complete.

We expose two APIs in the HClib C++ namespace: `Isend` and `Irecv`. These APIs have similar signatures to their MPI analogues. The API signature of `Isend` and `Irecv` is given below in Listing 1. The corresponding MPI routines are also listed in the comments for reference.

Listing 1. Communication API signature.

```
1
2  void Isend(COMMUNICATION_OBJ *data, int dest, int64_t tag,
3   hclib::promise_t<COMMUNICATION_OBJ*> *prom, MPI_Comm comm)
4
5  //MPI_Isend(void *buf, int count, MPI_Datatype datatype,
6  // int dest, int tag, MPI_Comm comm, MPI_Request *request)
7
8  void Irecv(int count, int source, int64_t tag,
9  hclib::promise_t<COMMUNICATION_OBJ*> *prom, MPI_Comm comm)
10
11 //MPI_Irecv(void *buf, int count, MPI_Datatype datatype,
12 //int source, int tag, MPI_Comm comm, MPI_Request *request)
```

We are targetting C++ rather than C, and therefore the benchmarks used C++ objects to represent data. Since MPI routines usually transport a block of contiguous memory, it is needed to convert these objects to a contiguous blob of data. Although MPI provides interfaces to create custom MPI datatype from a general set of datatypes, we felt it is more readable to extend the user's object with serialization/deserialization facilities. The first parameter of `Isend` is the data object that needs to be communicated. This object needs to be a subtype of the `hclib::communication::obj` type. The `hclib::communication::obj` is an abstract type with serialization and deserialization routines. One noticeable and important difference is the lack of data size parameter, which specifies the amount of data to be communicated. This size and blob of data to be transferred is obtained using the serialization of data which is explained in Section III-C1. The dest parameter is the destination rank, and the tag is the integer message tag as in MPI calls.

MPI non-blocking APIs use the `MPI_Request` parameter to signal the completion of the operation. Instead, we use a `promise`, the `prom` parameter, to signal the completion of the operation. Internally the runtime will do the job of putting the data into the `promise` to signal the completion of the communication operation.

`Irecv` uses a similar set of parameters but with a big difference in the first parameter. Here the first parameter specifies the number of bytes that need to be received. The runtime internally deserializes the bytes received to the required type requested by the user which is explained in Section III-C1.

A sample program using the communication APIs to transmit data across ranks is given below in Listing 2. Here we can see that the output promise `prom_recv` of `Irecv` at Line 32 is used as the task dependency in Line 39

Listing 2. An example program that performs basic send-receive communication.

```
1
2  class test_obj : public communication::obj {
3   public:
4     int n;
5
6     archive_obj serialize()
7     {
8       archive_obj ar_ptr;
9       ar_ptr.size = sizeof(int);
10      ar_ptr.data = malloc(ar_ptr.size);
11      memcpy(ar_ptr.data, &n, ar_ptr.size);
12      return ar_ptr;
13    }
14
15    void deserialize(archive_obj ar_ptr)
16    {
17      n = *(int*)(ar_ptr.data);
18    }
19 };
20
21 //Main Program
22
23 auto prom_recv = new hclib::promise_t<test_obj*>();
24 auto prom_send = new hclib::promise_t<test_obj*>();
25
26 if(rank == 0) {
27   auto *tst = new test_obj();
28   tst->n = 22;
29   communication::Isend(tst, 1, 1, 0, prom_send);
30 }
31 if(rank == 1) {
32   communication::Irecv(sizeof(int), 0, 1, prom_recv);
33
34   hclib::async_await( [=]()
35   {
36     auto recv_tmp = prom_recv->get_future()->get();
37     printf("Value Recv %d\n", recv_tmp->n);
38   },
39   prom_recv->get_future());
40 }
41
   Communication Operations
   Serialization/Deserialization Routines
   Task waiting for completion of communication
```

The communication APIs can be used directly within a non-resilient or resilient task without any special handling. However, there are some differences in their behavior. While using `Isend` inside a non-resilient task, the communication operation is immediately scheduled for execution. In a resilient

Non-resilient (Original API)	async_await(task_body_lambda, future_deps)
Replication	async_await_check(task_body_lambda, **promise_out**, future_deps)
Replay	async_await_check(task_body_lambda, promise_out, **error_check_fn**, future_deps)
ABFT	async_await_check(task_body_lambda, promise_out, error_check_fn, **ABFT_lambda**, future_deps)

TABLE I

RESILIENT APIS WITH THE ADDITIONAL PARAMETERS SHOWN IN RED COLOR COMPARED TO THE NON-RESILIENT VERSION. THE BOLD FONT EMPHASIZES THE NEW PARAMETER REQUIRED, COMPARED TO THE API JUST ABOVE THE ONE UNDER CONSIDERATION.

task, scheduling the communication is delayed until the error checking (or equality checking) succeeds. If we were to eagerly schedule communication and have the error checking fail, then we might be communicating the wrong values. Therefore we delay communication until error checking is successful. Similar delayed execution within resilient task applies to `Irecv` also.

1) Serialization and Deserialization: MPI generally supports sending of a contiguous block of memory. However, the data used by the user in the application may not be a contiguous block of memory, and therefore, it might become necessary to perform serialization when we want to communicate such data objects. One way is to ask the user to serialize the object and give the buffer as input to `Isend`. In that case, the destination rank that receives the buffer will need to perform the deserialization of the buffer to get back the usable data object. Since the `Irecv` operation is asynchronous, the user will need to perform the deserialization in a dependent task which waits for the completion of the communication operation. While converting from a single node program to its distributed version, this asynchronous usage of serialization/deserialization causes programming overheads.

To streamline this process, we decided to go ahead with using the typed user object directly while sending data and perform the serialization and deserialization within the extended HClib runtime which, in turn, passes the contiguous data buffer to MPI runtime. Thus the user needs to extend the `hclib::communication::obj` class to and create a custom data type to work on. Primarily the class needs to include a serialize routine that can convert the user object to a serialized object of type `archive_obj` and a deserialize that can give back the user object. The `archive_obj` includes the data buffer that represents the serialized user object and the size of the buffer as shown in Listing 3. A concrete example of serialization routines is shown in Listing 2 at lines 6 and 15.

Listing 3. Definition of Archive Object which is used for serialization to help perform MPI communication operations.

```
1 class archive_obj {
2   //size of object blob
3   int size = 0;
4
5   //blob of object to be archived
6   void *data = nullptr;
7 };
```

2) Process Crash Recovery: Section III-B discuss how to overcome fail-continue errors. The addition of the communication module to enable distributed execution also presents opportunities to overcome fail-stop errors. MPI-ULFM [16] is a fault-tolerant extension of MPI, that adds a small set of APIs thereby enabling users to implement application-specific recovery during MPI rank failures. Fenix [17], [18] exposes a user-friendly layer on MPI-ULFM to enable transparent recovery from process failures. Fenix provides mechanisms to use spare nodes to re-spawn the failed processes and restore the application state. Together, MPI-ULFM and Fenix enable the user to implement application-specific recovery during failures.

Adding application-specific recovery for each application is time-consuming and hurts programmer productivity. Therefore we wanted to remove this recovery process from the application programmer's workflow and automate it within the runtime. To achieve this, we presented our resilient runtime extension as the application to Fenix and MPI-ULFM. Our runtime extension keep tab of all messages send and received using the APIs in Listing 1 along with its completion status. Therefore, we can use this information to re-execute the communication operations in case of failure without user intervention.

Fenix allows us to restart the execution from the point of Fenix Initialization or continue execution on non-crashed ranks in case of a failure. Just re-executing all ranks from the point of Fenix initialization is a trivial recovery process. However, a better design would be to use the already computed partial results available in the non-crashed ranks rather than re-executing them. Based on this insight, our high-level design is to re-execute the full computation task graph only for the crashed rank and reuse the partial results on non-crashed ranks, thereby bypassing tasks that have already completed in them. This same approach is used in various runtimes such as Fenix and Reinit [22]. However, they require the user to manage the handling of error recovery.

As mentioned above, the crashed rank starts execution on a new spare rank, and therefore must execute the full task graph. As a result, it has to re-execute all communication operations that had been completed before the crash. This is required since all remote data received from neighbors before the crash will not be available in the new spare rank. To support the re-execution in spare rank, the non-crashed ranks will also have to participate in the corresponding communication operations. However, since the completed tasks are not re-executed in the non-crashed nodes in our design, in the absence of any other support, the user will have to add handling of these repeated communication operations as additional "compensation code".

However, we want to avoid adding this programming burden

on the user to achieve the re-execution of the completed communication operations and offload it to our runtime. To achieve this, we bookkeep all the send/receive operations and use this information to re-execute any required communication in the error handling callback provided by Fenix. In each rank, the runtime maintains a list of pending communication operations that the user has invoked. Once the runtime identifies a communication operation as completed using `MPI_Test`, the handle to this communication operation is moved from the pending list to a completed list. In the error handling callback, we reissue all operations on the pending list since the Fenix/MPI-ULFM runtime invalidates all outstanding communication operations during a failure.

The operations in the completed list can be classified into two categories, the ones to the crashed rank and others to the non-crashed ranks. We reissue all operations to the crashed rank since the spare rank executes all tasks from the start, including communication operations. From the completed operations to the non-crashed ranks, we can safely ignore the receive operations since the data is already available locally. From the send operations, we need to identify which among them needs to be reissued since `MPI_Isend` completion only means the send buffer can be reused and do not indicate the message is received. We tried to overcome this issue by using `MPI_Issend`, which waits for the matching receive to signal completion. But currently, MPI-ULFM is not well integrated with `MPI_Issend` and was behaving erroneously due to which we initially reissued all completed send operations.

Later we enhanced the protocol and introduced an all-to-all communication through which all nodes exchange the completed receive tags. Instead of reissuing all completed send operations, we ignore any send that matches these received tags. We use `MPI_Alltoallv` (currently not exposed the user) to exchange the tags. We believe once `MPI_Issend` is integrated with MPI-ULFM, we can find out the exact subset of send operations that needs to be reissued without the current all-to-all communication that we perform. Another point to note is that we do not free the serialized data buffer sent to remote ranks since the non-crashed nodes can reuse this serialized data buffers to send data in case of failure without recomputing. This can potentially cause scaling issues if many serialized data buffers get accumulated over a long period. In the future, we would also like to add occasional local checkpointing so that the spare rank can start from the local checkpoint rather than the initialization point. Once a local checkpoint of the current state is taken, we can free all serialized data buffers that were kept around as part of the compensation code. The crash tolerant version of the program shown Listing 2 is created by building it with an additional flag `-DUSE_FENIX` and the user does not have to add any compensation code anywhere in the program to enable recovery.

IV. Implementation

In this section, we discuss the implementation of our communication module prototype by extending the Habanero C++ library (HClib). An overview of HClib and its runtime capability are discussed in Section IV-A followed by efforts for the extension of HClib.

A. HClib

HClib [21] is a lightweight, work-stealing, asynchronous many-task based programming model based runtime. HClib focuses on offering simple tasking APIs with low overhead task creation. HClib is entirely library-based and supports both a C and C++ API and therefore does not require a custom compiler. HClib's runtime consists of a persistent thread pool, across which tasks are load balanced using lock-free concurrent deques. At the user-visible API level, HClib exposes several programming constructs that helps the user to express parallelism easily and efficiently.

A brief summary of the relevant APIs is below:

1) `launch`: Initialize the HClib runtime, including spawning runtime threads.
2) `async`: Dynamic, asynchronous task creation.
3) `finish`: Bulk, nested task synchronization. Waits on all tasks spawned within a given scope.
4) `promise` and `future`: Point-to-point inter-task synchronization. A promise is a single-assignment, thread-safe container that stores some value and a future is a read-only handle on that value. Waiting on a future causes a task to suspend until the corresponding promise is *satisfied* - i.e. some value is put to the promise.
5) `async_await`: Data-driven execution for tasks using `future` as task dependency

Also, HClib accepts a target platform model which is an abstraction of the homogeneous/heterogeneous hardware resources across which the application execution will be distributed. With this feature enabled, users can use the `async_await_at([] { body; }, dependency, place)` API to create a data-driven task at a specific *place*. More details can be found in [21]. In this paper, we use the `async_await_at` feature to offload asynchronous communication tasks onto dedicated communication workers.

B. Enabling Communication in Resilience Module

As mentioned in Section III-A the computation worker offloads all communication calls to communication workers. In the current implementation, we use one communication worker in each rank and hence configure MPI in the thread funneled mode.

As mentioned in Section III-C we allow the use of communication APIs throughout the program i.e., both non-resilient or resilient tasks can invoke these APIs without any special handling. However, the semantics of the communication calls on non-resilient vs. resilient versions are not the same.

For `Isend`, while using in a non-resilient task, the communication operations are immediately scheduled for execution. But when resilient tasks are used, the communication operations only get scheduled for execution after error checking succeeds. This delay is used to make sure error data is not communicated thereby removing the necessity to a rollback

of messages. We use task local storage to collect all communication operations invoked within a resilient task. Once the error checking succeeds we get back the list of communication operations saved to the task local storage and schedule them.

For `Irecv`, the distinction between resilient and non-resilient task is more subtle. Since we are receiving data from a different rank, there is no need to wait until error checking, which works on local data to succeed. Therefore any task can go ahead and schedule the communication operations. The only difference between resilient and non-resilient task is that resilient tasks might involve the creation of multiple replica tasks and therefore the runtime picks one task to schedule the MPI communication.

The `Isend` and `Irecv` operations notify completion using a promise rather than an `MPI_Request` object as in MPI. This difference implies we need a mechanism to convert `MPI_Request` returned by an MPI call to a promise. This is done through a combination of interactions between the communication API and the underlying work-loop in the communication worker.

When the user invokes an `Irecv` call, it creates a communication task to perform the actual MPI call. Within the communication task, we create a serialized object of type `archive_obj` with size as given in the `Irecv` call. We invoke the MPI operation with the data buffer from the serialized object. The MPI call returns an `MPI_Request` to query for status or completion of the operation. We add the `MPI_Request` along with the serialized object and the output promise parameter to a pending queue which gets polled by the communication worker loop.

The communication worker loop does periodical polling of the pending queue. It iterates over the list of pending MPI operations from the queue and process any that have completed. During processing, it first deserializes the received data (in case of `Irecv`) and satisfies the output promise associated with that communication operation. If there are no pending MPI operations, it yields so that other tasks can use the worker before polling again.

During failures, Fenix performs recovery and restarts execution from the point of Fenix initialization. This involves performing a `longjump` to the initialization point. The jumping across stack caused by `longjump` was interfering with cleanup procedures within the HClib tasking runtime. To overcome this issue, we added a new Fenix mode to continue the execution after recovery without getting back to the Fenix initialization point. Our recovery procedure, which performs the re-execution of communication, is added to the error handling callback invoked by Fenix during the recovery process.

V. Evaluation

This section presents the results of an empirical evaluation of our runtime system on a multi-node platform.

Purpose: Our goal is 1) to demonstrate that our programming model supports various resilience techniques with asynchronous MPI communication, and 2) to study the performance impact of the resilient runtime system.

Machine: We present the results on the Cori supercomputer located at NERSC, in which each node has two sockets, each of which has 16-core Intel Xeon E5-2698 v3 CPUs at 2.30GHz.

Benchmark: Our first benchmark is the stencil 1D benchmark that solves linear advection (a hyperbolic PDE): We break the 1-D domain up into contiguous, non-overlapping subdomain for inter-node parallelism. Each subdomain is further divided into contiguous, non-overlapping tiles, each of which will be time-advanced by an independent task on each iteration to allow for intra-node parallelism. Time-advancing the left-most value on each tile requires the right-most previous value from the left neighbor by performing intra/inter-node communication. Similarly, time-advancing the right-most value requires the left-most previous value from the right neighbor. For this reason, a task will be ready to execute once the tasks that generate the previous data for its own tile and the neighboring tiles to the left and right have completed. In each node, we use 128 tiles of size 16,000 doubles, 128 time steps per iteration (each task advances its assigned tile 128 time steps), and 4,096 iterations.

Our second benchmark is the Smith-Waterman algorithm that performs local sequence alignment, which is widely used for determining similar regions between two strings of nucleic acid sequences. In each node, use two input strings of sizes 167,040 and 172,800, divided among 4,096 tiles arranged as 64×64.

Experimental variants: The benchmark was evaluated by comparing the following versions:

- **Baseline:** Non-resilient execution with `async_await` with the asynchronous communication API (`communication::Isend`, `communication::Irecv`).
- **Replay:** Resilient execution with `replay::async_await_check` with the asynchronous communication API.
- **Replication:** Resilient execution with `replication::async_await_check` with the asynchronous communication API.

To the three versions listed above, we also add variants where we crash a process to check our runtime's tolerance to fail-stop errors and use the support from Fenix and MPI-ULFM to recover.

Runtime Settings: We used the HClib-based resilient runtime system [23] described in Section III and IV for this evaluation. We used GCC 8.3.0 and MPI-ULFMv4.0.2u1 for compiling and running benchmarks as well as the HClib runtime extensions.

We measured weak scaling performance using up to 9 nodes with an additional spare rank (i.e. 10 nodes in total). We start the execution with an additional spare rank rather than dynamically spawning during a crash because Fenix currently does not support a new spare rank's dynamic creation. Our runtime does not depend on when the spare is created and

can work with both scenarios. Also, we used 32 workers per node. The performance was measured in terms of elapsed seconds from the start of the parallel computation(s) to their completion.

A. Performance Numbers without failures

To show the overhead of the distributed resilient runtime, we measured the execution time of the Stencil1D benchmark using task replay and task replication resilience techniques without failures. Figure 2 shows the weak-scaling numbers with up to nine nodes. The graph shows good scalability since the time taken for each variant across the various number of nodes remains almost constant.

Task Replay vs. Non Resilient (Baseline) The performance of task replay is comparable to the non-resilience variant, which means our implementation is efficient. The small overhead comes from the additional error checking needed for replay and the necessary bookkeeping to collect promises with the task so that their signaling can be delayed until error checking succeeds.

Task Replication vs. Non Resilient (Baseline) The performance of task replication is approximately 2× slower than the non-resilience variant because the replication variant executes each task two times.

Analyses of Communication Overheads

The results show good scalability for the stencil benchmark. This is because, for each time step, all tiles (or tasks) within a node can be executed independently, and similarly, all nodes can also be executed simultaneously. This parallel execution is possible because there are no dependencies within a timestep. To get more insight into the results we analyzed them using HPCToolkit ([24], Fig. 3), which enables sampling-based performance measurements with low-overhead, on Cori. With the HPCToolkit using a sampling frequency of 1kHz, we ran the 1D stencil code with the same data size on 4 ranks and collected its profile using the total time (`REALTIME`) event. Table II includes the performance counter numbers for `libmpi.so` in the Non Resilient, Task Replay, and Task Replication variants. Note that we report the inclusive metric in Table II, which means the cycles contains the value measured for `libmpi.so` itself as well as costs incurred by any functions it calls. The results show that the communication part only accounts for 1.2-2.0% of the whole program execution, and most of the time is spent in the 1D stencil user program (95%), which is why we saw great weak scaling numbers. Table II also shows that the percentage of communication is low on the replication variant compared to the other two. This is because the kernel computation is doubled in the replication variant since it executes the kernel two times, but the communication remains the same. This increase in kernel execution time gets reflected as a lower communication percentage.

For the smith-waterman benchmark, we see that the execution time increases with the number of nodes. This is because of the dependency that is present in the smith-waterman algorithm where tile[i,j] depends on tiles tile[i-1,j], tile[i,j-1] and tile[i-1,j-1]. Due to this dependency, nodes that contain tiles with higher values of i and j have to wait when their dependencies are calculated in some earlier nodes. We did not try to optimize this part since our focus was on incorporating resilience, given a parallel program rather than optimizing the parallel program itself.

	`Communication(%)`
Baseline	2.0%
Task Replay	1.9%
Task Replication	1.2%

TABLE II
THE SAMPLED REALTIME % NUMBERS FOR `LIBMPI.SO` ON CORI.

B. Performance Numbers with Fail-continue Errors

To check the effectiveness of our resilience mechanisms in the presence of soft errors, we ran the benchmarks while injecting failures. We inject failures deterministically by inputting a list of specific tasks that should fail and then flipping a deterministic bit in the tile or cube output from that task. The bit was carefully chosen to ensure no false positives or false negatives with checksums. Note that this is not meant to model real failure behavior. Each task was independently given some probability of failure in the 0.01% to 1% range. Under replay resilience, we allowed at most two failed attempts per unique task because that was the runtime limit; increasing this limit would have increased the memory overhead of replay resilience. We allowed at most one replica to fail under replication resilience because triplication requires two out of three replicas to agree. Under these constraints, we built up our list of failed tasks by evaluating whether each individual task should fail.

Figure 4 shows how the failure injections degrade the performance of the replay and replication runtimes. We injected errors at a rate of 1% and 10% per node. Here, X% implies that an error is injected into X% of the total tasks per node. The results show that the increase in execution time closely follows the amount of failure occurred. The performance degradation is proportional, for example around 10% with the 10% failure rate case, which is an extremely high rate in real-world systems [25].

C. Performance Numbers with Fail-stop Errors and/or Fail-continue Errors

To check the effectiveness of our resilience mechanisms in the presence of hard failures, we ran the benchmarks while crashing a process. To accomodate the process crash, execution is started with a spare rank in reserve. We crash a process at two points in all the three variants, namely baseline, replay, and replication. One experimental variant introduces a crash towards the beginning of execution, and another variant crashes towards the middle of the execution. To add fail-continue errors, each task is also independently given some probability of failure in the 1% range by introducing soft errors to the replay and replication variants. This helps us to demonstrate that our runtime can tolerate both soft and hard failures

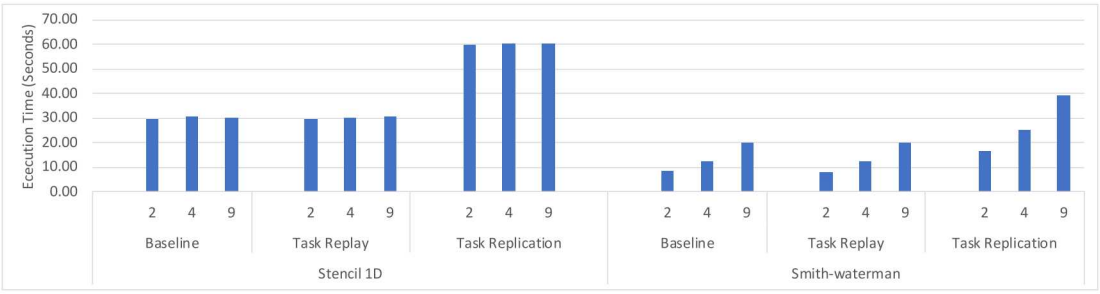

Fig. 2. Weak scaling of Stencil 1D and Smith-waterman with different number of nodes (2, 4 and 9) and resilience schemes without fault injection.

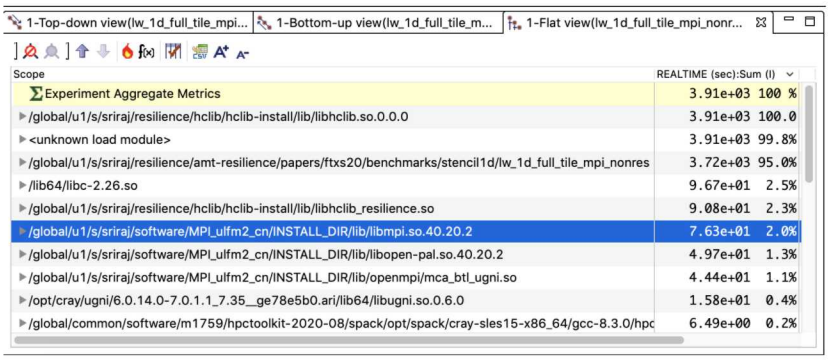

Fig. 3. HPCToolkit showing the percent of execution time used by various components of the program. The stats show that most of the time (95%) is spent in the 1D-stencil user program, and the overheads due to HClib and communication is minimal.

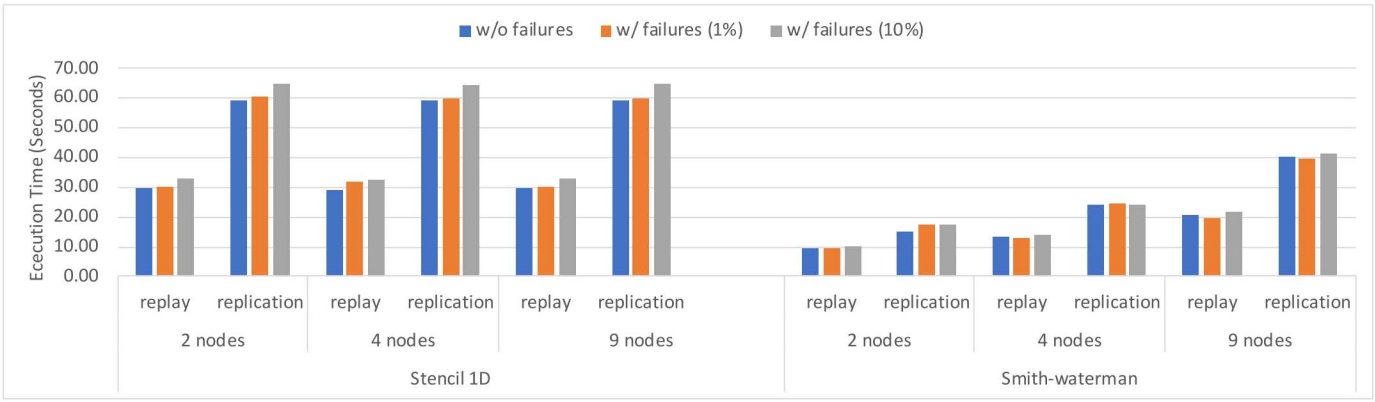

Fig. 4. Comparison of performance degradation in different resilience schemes with soft errors injected at 1% and 10% rates.

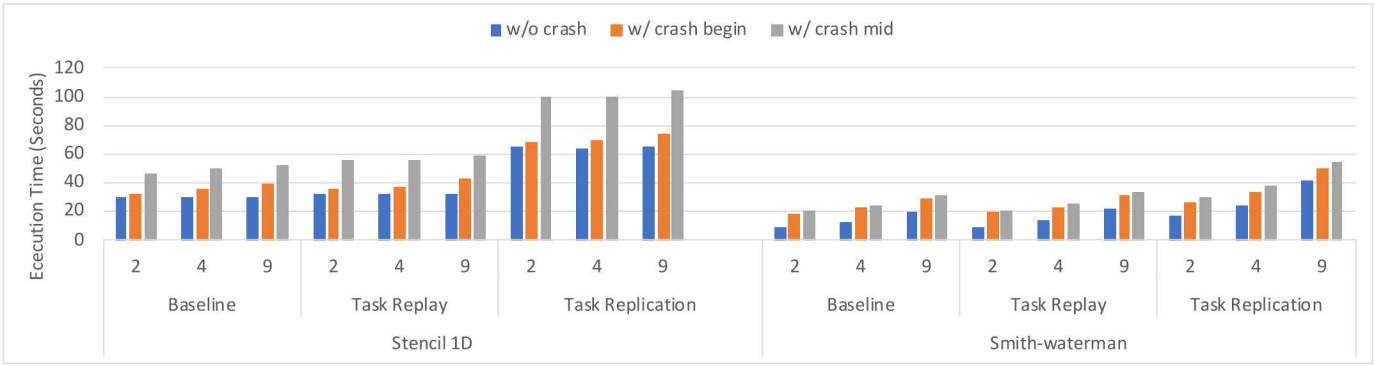

Fig. 5. Comparison of performance degradation in different resilience schemes with multiple nodes (2, 4 and 9) along with hard failure such as process crash.

together. We used a timeout (-mca mpi_ft_detector_timeout parameter) of 3 sec for the 2-node case, 5 sec for the 4-node case, and 10 sec for the 9-node case for the stencil 1D experiment. The timeout values that can tolerate errors for the given node count were empirically found by running the experiments. In Figure 5, the timeout's impact can be noticed in the recovery time for early crash (blue vs. orange), which is very low for the 2-node case, but higher for the 9-node case. For the smith-waterman benchmark, we use a timeout value of 10 sec for all node counts. The smith-waterman benchmark has a lower amount of communication compared to the stencil 1D benchmark. Using a small timeout might result in MPI-ULFM accidentally report as an error due to the smith-waterman benchmark's infrequent communication patterns.

We can see that recovering from a crash increases the time by around 10 seconds for smith-waterman in Figure 5. Also, Fenix/MPI-ULFM runtime takes around 10 seconds to recover in case of a crash in the current experimental setting with the given timeout. The same pattern can be seen for the stencil 1D benchmark, such as the 2-node case taking only 3 seconds to recover. Thus we can see that our communication runtime does not add much overhead compared to what we can achieve using Fenix and MPI-ULFM for recovering from a failure. More importantly, our runtime achieves this recovery without any compensation code from the user, thereby improving programmer productivity.

VI. Related Work

There is an extensive body of literature on software-based resilience scheme for SPMD programs, including coordinated checkpoint and restart (C/R). However, enabling resilience in the AMT programming model has not yet been well studied despite the fact that the abstraction of tasks and data in AMT models facilitates modeling recovery patterns to enable asynchronous and localized application recovery with simplicity. For example, Subasi *et al* [26], [27] study a combination of task replication/replay and checkpoint/restart for a task-parallel runtime, OmpSs [11]. Cao *et al* [28] has a similar replay model in the PaRSEC task-based runtime framework. However, these approaches do not support resiliency across multiple nodes.

In the context of MPI+X, where X is an AMT programming model, one of our prior work, HiPER [21] (a.k.a HClib), offers an MPI module that enables the composability of MPI APIs with an AMT programming model. However, HiPER does not support resiliency, and this paper provides a resilience module with the HiPER runtime.

We used MPI-ULFM [16] and Fenix [18] to enable global recovery and restart. Reinit [22], [29] is another interface that supports global-restart recovery and provides an easier interface compared to MPI-ULFM. Reinit requires modifications to the job scheduler, which makes it hard to deploy. Reinit++ [30] is a new design and implementation of the Reinit approach for global-restart recovery that removes the need for application re-deployment. We used MPI-ULFM/Fenix as our recovery interface due to our current familiarity with them. Our runtime can be interface with any alternatives such as Reinit++, which can enable MPI recovery.

In summary, our approach provides a unified AMT programming model that enables resiliency across multiple nodes by enabling a unified execution of resilient, non-resilient, and communication tasks in a single framework to accommodate fail-stop and fail-continue errors.

VII. Conclusions and Future Work

Resilience in bulk-synchronous MPI programming model was supported using the traditional checkpoint/restart (C/R) approach. This causes a disproportionate use of resources by triggering global recovery even for local failures. Also, this model worked under the assumption that a fault is a rare event. The Asynchronous Many-Task programming model decomposes the program into a set of tasks, thereby removing the need for bulk synchronization. The task decomposition also allows localization and isolation of faults in the program thus making a recovery scalable and inexpensive. Therefore by composing MPI with an AMT runtime we are able to enable scalable localized fault tolerance with minimal overhead. Additionally, in case of hard failures such as a crash, we use the recovery mechanisms provided by MPI to recover transparently without user intervention, thereby enhancing productivity. During this exercise, we also identified that MPI routines such as `MPI_Issend` can aid in the user's recovery process and hence needs to be supported in MPI-ULFM.

In the future, we would like to incorporate more communication routines other than send and receive. We want to experiment with tuning error detection using other mechanisms, for example, `mpi_ft_detector_thread` instead of the timeout. We would also like to add occasional local checkpointing so that the spare rank can start from the local checkpoint rather than the initialization point in case of a failure. We would also like to try with multilevel checkpointing that can, in turn, help with creating multilevel resilient tasks. We would also like to study the characteristics of faults in real-world systems and use error rates and patterns accordingly. We would also like to extend our implementation to support recovery from multiple node failures.

Artifact Availability

https://github.com/srirajpaul/hclib/tree/feature/resilience_mpi/modules/resilience

Acknowledgment

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration (NNSA) under contract DE-NA0003525. This work was funded by NNSA's Advanced Simulation and Computing (ASC) Program. This paper describes objective technical results and analysis. Any subjective views or opinions that might be expressed in the paper do not necessarily represent the views of the U.S. Department of Energy or the United States Government.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility operated under Contract No. DE-AC02-05CH11231.

REFERENCES

[1] C. D. Martino *et al.*, "Measuring and understanding extreme-scale application resilience: A field study of 5,000,000 hpc application runs," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, June 2015, pp. 25–36.

[2] A. Moody *et al.*, "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, 2010, pp. 1–11.

[3] P. H. Hargrove and J. C. Duell, "Berkeley lab checkpoint/restart (blcr) for linux clusters," *Journal of Physics: Conference Series*, vol. 46, no. 1, p. 494, 2006. [Online]. Available: http://stacks.iop.org/1742-6596/46/i=1/a=067

[4] F. Cappello, K. Mohror, B. Nicolae, R. Gupta, S. Di, A. Moody, E. Gonsiorowski, and G. Becker, "VeloC," https://veloc.readthedocs.io/en/latest/, 2018.

[5] J. Chung *et al.*, "Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 58:1–58:11.

[6] H. Kaiser *et al.*, "Parallex an advanced parallel execution model for scaling-impaired applications," in *2009 International Conference on Parallel Processing Workshops*, 2009, pp. 394–401.

[7] C. Augonnet *et al.*, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, pp. 187–198, 2011.

[8] M. Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12, 2012, pp. 66:1–66:11.

[9] G. Bosilca *et al.*, "PaRSEC: Exploiting Heterogeneity to Enhance Scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[10] A. Duran *et al.*, "Ompss: a proposal for programming heterogeneous multi-core architectures," *Parallel Processing Letters*, vol. 21, no. 2, pp. 173–193, 2011.

[11] A. Fernández *et al.*, "Task-Based Programming with OmpSs and Its Application," in *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part II*, 2014, pp. 601–612.

[12] J. Bennett *et al.*, "ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platform," Sandia National Laboratories, Tech. Rep. SAND2015-8312, September 2015.

[13] T. G. Mattson *et al.*, "The open community runtime: A runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, Sept 2016, pp. 1–7.

[14] S. R. Paul, A. Hayashi, N. Slattengren, H. Kolla, M. Whitlock, S. Bak, K. Teranishi, J. Mayo, and V. Sarkar, "Enabling resilience in asynchronous many-task programming models," in *Euro-Par 2019: Parallel Processing - 25th International Conference on Parallel and Distributed Computing, Göttingen, Germany, August 26-30, 2019, Proceedings*, 2019, pp. 346–360.

[15] F. Cappello *et al.*, "Toward exascale resilience: 2014 update," *Supercomput. Front. Innov.: Int. J.*, vol. 1, no. 1, pp. 5–28, Apr. 2014.

[16] W. Bland, A. Bouteiller, T. Herault, G. Bosilca, and J. J. Dongarra, "Post-failure recovery of MPI communication capability: Design and rationale," *International Journal of High Performance Computing Applications*, p. 1094342013488238, 2013.

[17] M. Gamell, K. Teranishi, E. Valenzuela, M. Heroux, and M. Parashar, "Fenix, a fault tolerant programming framework for mpi applications, version 00," 10 2016. [Online]. Available: https://www.osti.gov//servlets/purl/1336604

[18] M. Gamell, D. S. Katz, H. Kolla, J. Chen, S. Klasky, and M. Parashar, "Exploring automatic, online failure recovery for scientific applications at extreme scales," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2014, pp. 895–906.

[19] B. Bouteiller, F. Cappello, T. Herault, K. Krawezik, P. Lemarinier, and M. Magniette, "MPICH-V2: a Fault Tolerant MPI for Volatile Nodes based on Pessimistic Sender Based Message Logging," in *Supercomputing, 2003 ACM/IEEE Conference*, Nov 2003, pp. 25–25.

[20] H. C. Edwards and B. A. Ibanez, "Kokkos' Task DAG Capabilities," Sandia National Laboratories, Tech. Rep. SAND2017-10464, September 2017.

[21] M. Grossman *et al.*, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 723–732.

[22] I. Laguna, D. F. Richards, T. Gamblin, M. Schulz, B. R. de Supinski, K. Mohror, and H. Pritchard, "Evaluating and extending user-level fault tolerance in mpi applications," *International Journal of High Performance Computing Applications*, vol. 30, no. 3, 1 2016.

[23] "HClib Resilience Branch," https://github.com/srirajpaul/hclib/tree/feature/resilience_mpi, accessed: 2019-09-03.

[24] L. Adhianto *et al.*, "Hpctoolkit: Tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, no. 6, pp. 685–701, Apr. 2010. [Online]. Available: http://dx.doi.org/10.1002/cpe.v22:6

[25] M. Gamell, D. S. Katz, K. Teranishi, M. A. Heroux, R. F. Van der Wijngaart, T. G. Mattson, and M. Parashar, "Evaluating Online Global Recovery with Fenix Using Application-Aware In-Memory Checkpointing Techniques," in *45th International Conference on Parallel Processing Workshops (ICPPW)*. IEEE, 2016, pp. 346–355.

[26] O. Subasi *et al.*, "Designing and modelling selective replication for fault-tolerant hpc applications," in *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, May 2017, pp. 452–457.

[27] ——, "Nanocheckpoints: A task-based asynchronous dataflow framework for efficient and scalable checkpoint/restart," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pp. 99–102.

[28] C. Cao *et al.*, "Design for a soft error resilient dynamic task-based runtime," in *2015 IEEE International Parallel and Distributed Processing Symposium*, May 2015, pp. 765–774.

[29] S. Chakraborty, I. Laguna, M. Emani, K. Mohror, D. K. Panda, M. Schulz, and H. Subramoni, "Ereinit: Scalable and efficient fault-tolerance for bulk-synchronous MPI applications," *Concurr. Comput. Pract. Exp.*, vol. 32, no. 3, 2020. [Online]. Available: https://doi.org/10.1002/cpe.4863

[30] G. Georgakoudis, L. Guo, and I. Laguna, "Reinit^{++}: Evaluating the performance of global-restart recovery methods for MPI fault tolerance," in *High Performance Computing - 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22-25, 2020, Proceedings*, ser. Lecture Notes in Computer Science, P. Sadayappan, B. L. Chamberlain, G. Juckeland, and H. Ltaief, Eds., vol. 12151. Springer, 2020, pp. 536–554. [Online]. Available: https://doi.org/10.1007/978-3-030-50743-5_27