SAND2019-7376C

Enabling Resilience in Asynchronous Many-Task Programming Models *

Sri Raj Paul[1], Akihiro Hayashi[2], Nicole Slattengren[3], Hemanth Kolla[3], Matthew Whitlock[3], Seonmyeong Bak[1], Keita Teranishi[3], Jackson Mayo[3], and Vivek Sarkar[1]

[1] Georgia Institute of Technology, Atlanta, Georgia, USA
{sriraj, sbak5, vsarkar}@gatech.edu
[2] Rice University, Houston, Texas, USA, ahayashi@rice.edu
[3] Sandia National Laboratories, Livermore, California, USA
{nlslatt, hnkolla, mwhitlo, knteran, jmayo}@sandia.gov

Abstract. Resilience is an imminent issue for next-generation platforms due to projected increases in soft/transient failures as part of the inherent trade-offs among performance, energy, and costs in system design. In this paper, we introduce 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. Our approach makes it possible for the application programmer to declaratively express resilience attributes with minimal code changes, and to delegate the complexity of efficiently supporting resilience to our runtime system. We have created a prototype implementation of our approach as an extension to the Habanero C/C++ library (HClib), where different resilience techniques including task replay, task replication, algorithm-based fault tolerance (ABFT), and checkpointing are available. Our experimental results show that task replay incurs lower overhead than task replication when an appropriate error checking function is provided. Further, task replay matches the low overhead of ABFT. Our results also demonstrate the ability to combine different resilience schemes. To evaluate the effectiveness of our resilience mechanisms in the presence of errors, we injected synthetic errors at different error rates (1.0%, and 10.0%) and found modest increase in execution times. In summary, the results show that our approach supports efficient and scalable recovery, and that our approach can be used to influence the design of future AMT programming models and runtime systems that aim to integrate first-class support for user-level resilience.

* 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.

Keywords: Resilience · AMT Runtimes · Habanero C/C++

1 Introduction

High performance computing plays a critical role in the advancement of science and engineering through simulations of large complex systems. Due to the insatiable demand for increased computing capability, multiple nations have committed to the development of exascale supercomputers. One of the major new challenges in exascale computing is the projected increases in silent data corruptions (SDC) [7], which are unexpected alterations in computation or data that can occur undetected. In such cases, application and software level mechanisms can play an essential role in improving application resilience.

The most popular resilience technique for application users today is coordinated checkpoint and restart (C/R) typically with bulk-synchronous parallel programming models [21], which involves global coordination of processing elements (PEs) for identifying a consistent global application state. However, this global recovery model is better suited for hard failures, and suffers from excessive performance overheads when global recovery is triggered for transient local failures. However, a majority of application failures are attributed to local node/process failure as reported by [21], with the recognition that recovery can potentially be applied only to the corrupted processes and data without requiring global coordination. Another example of local recovery is Containment Domains (CDs) [8], which provide an abstraction of error detection and correction to a local boundary intended for efficient and transparent recovery of HPC applications.

Asynchronous many-task (AMT) programming models [2–5, 9, 17, 20] are intended for managing the increasing complexity of node architectures and heterogeneity. 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 the application at the process level (MPI ranks). 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. We believe that the AMT foundations of transferable units and dynamic load balancing are also conducive to supporting fault tolerance. Specifically, we claim that AMT models are better suited to enabling local error recovery in next-generation platforms than bulk-synchronous models, since AMT models provide explicit abstractions of data and tasks, i.e., 1) a task represents a small piece of program execution, 2) failures are manifested as failed or lost tasks, and 3) failures can typically be remediated using lightweight mechanisms such as task replay.

In this paper, we introduce a comprehensive approach to enabling resilience in AMT programming models. While some of the prior approaches discuss different resilience techniques including task replay, task replication, algorithm-based fault tolerance (ABFT [15]), and checkpoint/restart for different AMT program-

ming models such as OmpSs [24] and PaRSEC [6], they are usually limited to a specific technique or to a specific application domain. Our approach complements existing checkpoint/restart mechanisms with reusable APIs to enable abstraction of data and program execution to map to multiple resilience patterns, and compositions thereof. To the best of our knowledge, this is the first work to discuss the design, implementation, and evaluation of a unified programming model that supports multiple resilience techniques.

Specifically, this paper makes the following contributions:

1. Programming model extensions to enable resilience techniques from past work (replay, replication, ABFT, checkpoint/restart) for AMT applications.
2. Support for arbitrary compositions of the extensions in 1.
3. Unified execution of resilient and non-resilient tasks in a single framework.
4. Implementation of our approach as extensions to the Habanero-C/C++ library for many-task parallelism.
5. Comprehensive performance evaluation of our implementation with synthetic error rates, and analysis of the results.

2 Design

A key question for any resilience design is to identify a program location at which we can perform error checking and recovery. For AMT programming models, the task boundary provides an ideal location around which resilience can be implemented. The task constructs that are of our interest do not involve internal synchronization, i.e., once a task is started, it runs to completion without blocking or waiting for other tasks or data. This implies that a task can start only after it acquires all its inputs, and that we can publish the results once it is finished; therefore, the task boundary provides a natural fit as the location around which resilience can be implemented, without worrying about internal task states or the global application state.

Once the program location around which resilience can be implemented is identified, the next step is to identify the data that needs to be checked to ensure correctness. A trivial choice is to ensure the integrity of the entire data used in the program, but this could be very expensive to implement and also unnecessary. The next obvious choice is to look at data that is going to be used past the task boundary, i.e. the task outputs. It is common for task-based runtimes to discourage the use of global variables for communicating data between tasks, and instead use built-in constructs for task inputs and outputs. For example, Legion [3] uses Logical regions, and Open-Community-Runtime(OCR) [20] uses Data-blocks to share data between tasks. C++11 includes `promise` and `future` constructs to enable transfer of data between tasks along with synchronization to avoid data races. A `promise` is a thread-safe container that uses single-assignment semantics to fill its value. The filled value can be read using a read-only handle called a `future`. `promise` and `future` together enable point-to-point synchronization between one source task to many sink tasks. Thus, if

the application programmer uses only `promise-future` pairs to perform communication between tasks, then the data in the `promise` objects becomes the live data at the task boundaries. Thus, we have identified both the program location and data that needs to be checked to enable resilience for applications based on AMT runtimes. Errors in the global state can be handled by other global recovery approaches; our approach is still beneficial in such cases because its support for local recovery for tasks enables more scalable and efficient resilience relative to the use of global recovery everywhere.

We assume, that for the same inputs, the task generates promises with data that is within some known range. Tasks do not need to be entirely deterministic - random numbers, etc. can be used within tasks so long as errors within the margin of the randomization's effect are permissible.

2.1 Resilient API Specifications

To reiterate, the key components of our approach to enable resilience in AMT runtimes are tasks and promise/futures. This section discusses our resilient API design. In short, we identified appropriate software abstractions that allow programmers to easily enable/disable different resilience techniques while keeping the original program mostly unchanged. In the following listings, we use `async` as a generic construct that creates an asynchronous task with a user-provided lambda expression, and `async_await` is a variant of `async` that can wait on one or more futures.

First, as a baseline implementation without resilience, Listing 1.1 shows a code example where the function `operation_val()` in Line 17 creates an asynchronous task waiting on the completion of two tasks, namely `read_first_val()` and `read_second_val()`. As shown in Lines 8 and 14, a `future` is satisfied by performing a `put` operation on the corresponding `promise`. Once the promises are satisfied, the `operation_val()` task which depends on the two promises is scheduled for execution. After the completion of the task, the result (`res`) is printed in the `print_result()` task.

Listing 1.1. A baseline non-resilient AMT program to perform an operation on two asynchronously generated values.

```
1  auto val1_dep = new promise();
2  auto val2_dep = new promise();
3  auto res_dep = new promise();
4
5  void read_first_val() {
6    async([=] {
7      val1 = new value(get_val_from_src());
8      val1_dep->put(val1);
9    }); // async
10
11 void read_second_val() {
12   async([=] {
13     val2 = new value(get_val_from_src());
14     val2_dep->put(val2);
15   }); // async
16
17 void operation_val() {
18   async_await([=] {
19     val1 = get_value(val1_dep);
20     val2 = get_value(val2_dep);
21     res = new value(op(val1, val2));
22     res_dep->put(res);
23     },val1_dep->get_future()
24      ,val2_dep->get_future());//async
25 }
26
27 void print_result() {
28   async_await([=] {
29     res = get_value(res_dep);
30     print(res);
31   },res_dep->get_future()); // async
32 }
   Task Creation
   Satisfying a promise
   Waiting on a promise
```

Replication Task replication is aimed at proactive reliability enhancement by executing the same task multiple times, assuming that the majority of the replicas produce the same output for determining correctness. The obvious drawback is the increase in computational cost, but it is still effective in situations where a few tasks in a critical path of the task graph may leave the computing system underutilized. The replication overhead can be reduced by selective replication to control the trade-offs between reliability and performance penalties.

Since task replication is based on equality checking of the outputs of the replica tasks, the runtime can internally take care of performing the replication and equality checking. There is no need for the user to provide any additional information other than the equality checking operator for each data type used. Also, the task APIs should include a mechanism to communicate the result of equality checking. This can be done using a promise that will have a value of 1 for success and 0 for failure. The replication version of the `operation_val()` task from Listing 1.1 is shown Listing 1.2. We can see that the only modification required in user code is to change the name of the task creation API and add a parameter, the `err_dep` promise which tells whether a majority of the replicas produced the same output.

The only data that gets propagated to dependent tasks are those that are `put` to a `promise`. With non-resilient tasks, dependent tasks get scheduled for execution once the necessary `put` operations have been performed. In order to prevent errors discovered in replication from propagating to dependent tasks, we do not publish any `put` operations from a replicated task until the equality checking of the replicas succeed.

Listing 1.2. Resilient AMT program based on replication to perform an operation on two asynchronously generated values.

```
1  auto err_dep = new promise();
2
3  void operation_val() {
4    replication::async_await_check([=] {
5      val1 = get_value(val1_dep);
6      val2 = get_value(val2_dep);
7      res = new value(op(val1, val2));
8      res_dep->put(res)
9    },err_dep,
10     val1_dep->get_future(),
11     val2_dep->get_future());//async
12 }
13
14 void print_result() {
15   async_await([=] {
16     recoverable = get_value(err_dep);
17     if (recoverable == 0) exit(1);
18     res = get_value(res_dep);
19     print(res);
20   }, res_dep->get_future(),
21      err_dep->get_future()); // async
22 }
```

The task replication construct
A promise with a failure status

Listing 1.3. Resilient AMT program based on replay to perform an operation on two asynchronously generated values.

```
1  bool err_chk_func (void *data) {
2    if (data is good) return true;
3    else return false;
4  }
5
6  auto err_dep = new promise();
7  void *chk_data = nullptr;
8
9  void operation_val() {
10   replay::async_await_check}([=]{
11     val1 = get_value(val1_dep);
12     val2 = get_value(val2_dep);
13     res = new value(op(val1, val2));
14     res_dep->put(res);
15     chk_data = res;
16   }, err_dep,err_chk_func, chk_data,
17   val1_dep->get_future(),
18   val2_dep->get_future()); // async
19 }
```

The task replay construct
User-defined error checking function
Arguments to the error checking

Replay Task replay is a natural extension of Checkpoint/Restart for the conventional execution models. Instead of applying a rollback of the entire program, as few as one tasks are replayed when an error is detected. Task replay is more sophisticated than replication but has much less overhead. In this form of resilience, the task is replayed (up to N times) on the original input if its execution resulted in some errors. Compared to replication, the application programmer needs to provide an error checking function so that the runtime can use it to check for errors. User-visible abstraction for a replay task is to extend the task creation API to include an error checking function and data on which that function operates. The application programmer needs to fill the data (`chk_data`) that needs to be checked for errors using the error checking function (`err_chk_func`). The replay version of the `operation_val()` task from Listing 1.2 is shown Listing 1.3. Similar to Replication tasks, Replay tasks also do not publish the output until error checking succeeds.

Algorithm-Based Fault Tolerance (ABFT) Algorithm-based fault tolerance (ABFT) mitigates failures using algorithm or application specific knowledge to correct data corruptions and computation errors. One of the seminal papers [15] introduced checksums that are embedded into the matrix and vector operators in parallel dense matrix computations to enable runtime error detection and correction. By using the numerical properties of the algorithm, ABFT uses checksums or provides alternative formulations to recover from an error thus ensuring forward progress without redoing the whole computation. Thus the API designed for an ABFT task should provide a facility to check for errors and if there is an error, a way to recover from it. Therefore, the user level abstraction to include ABFT is to extend the replay task API with a recovery facility as shown in Listing 1.4.

Listing 1.4. Resilient ABFT task signature containing the error correction mechanism.

```
1  abft::async_await_check([=] {
2    actual computation
3  }, err_dep, err_chk_func, chk_data,
4      err_correction_func, futures);

The ABFT construct
User-defined error correction function
```

Checkpoint/Restart (C/R) Checkpointing involves the saving of intermediate program state/outputs on to secure storage so that in case of failure, the application can be restarted from the point when the checkpoint was taken rather than from the beginning of the program's execution. From the context of task-based runtimes, once the error/equality checking succeeds at the boundary of a task, the output data can be checkpointed. Later in some following task, if all other resilience techniques fail, it can re-fetch the input data from the checkpoint and execute again.

Checkpointing can be added to any of the resilient tasks listed above. A proposed user level abstraction for a checkpoint task created by extending the

replay task is shown in Listing 1.5. The only addition is to specify where to store the checkpoint data using the `set_archive_store` API as shown in Line 13. In our current preliminary implementation, we keep a copy of the checkpoint in the memory itself, as in diskless checkpointing [22]. Efficiently performing these checkpoints and their performance evaluation is a topic for future work.

Listing 1.5. Resilient AMT program based on replay to perform an operation on two asynchronously generated values and also checkpoints the results.

```
1  void operation_val() {
2    checkpoint::async_await_check([=] {
3      val1 = get_value(val1_dep);
4      val2 = get_value(val2_dep);
5      res = new value(op(val1, val2));
6      res_dep->put(res);
7      chk_data = res;
8    }, err_dep, err_chk_func, chk_data,
9       val1_dep->get_future(),
10      val2_dep->get_future()); // async
11 }
12
13 set_archive_store(storage object);

   The checkpoint/restart construct
   The checkpoint API that specifies where to store the checkpoint data
   (invoked just once, before async_await_check)
```

2.2 Memory Management

C++ requires the user to perform memory management; i.e., the application programmer needs to explicitly free any data that is allocated in the heap memory. This could be reasonable to manage in normal AMT programs, but when we introduce resilience techniques manual deallocation poses certain challenges.

Many resilience techniques involve multiple executions of the task to get the correct results. This would mean that the user needs to keep track of the good or bad executions of the task. For the good runs, the data generated by a task would be used later in some consumer tasks; therefore, they need to be deallocated only after the consumption of the data. For bad runs, there is no need for the data created in the task and, therefore, they need to be deallocated at the end of the producer task itself. Keeping track of good or bad runs and selectively deallocating memory would create unnecessary complexity in the application code.

Therefore, to reduce the user's burden of manual memory management, we decided to add the reference counting capability that deallocates the data automatically once its use is over. Since data is being transferred between tasks using `promise` and `future`, reference counting is added by extending the `promise` to include the reference count. Ideally, the reference count specifies the number of tasks dependent on the `future` associated with the `promise`. The reference count is passed on to a promise when it is created. In other words, a reference count `N` specifies that only `N` tasks consume data from the given `promise`, and therefore the `promise` and the associated data can be freed once `N` tasks have used it.

3 Implementation

In this section, we discuss the implementation of our resilient-AMT prototype [1], extended from the Habanero C++ library (HClib) [11]. An overview of HClib and its runtime capability are discussed in Section 3.1 followed by efforts for the extension of HClib.

3.1 HClib

HClib [11] is a lightweight, work-stealing, task-based programming model and runtime that focuses on offering simple tasking APIs with low overhead [13] task creation. HClib is entirely library-based (i.e. does not require a custom compiler) and supports both a C and C++ API. HClib's runtime consists of a persistent thread pool, across which tasks are load balanced using lock-free concurrent dequeues. At the user-visible API level, HClib exposes several useful programming constructs. A brief summary of the relevant APIs is as follows. The `hclib::launch()` API initializes the HClib runtime, including spawning runtime threads. The `async([] { body; })` API creates a dynamic task executing `body` provided as a C++ lambda expression; this API optionally allows the inclusion of parameters that specify precondition events thereby supporting event-driven execution for tasks when so desired (i.e., the `async_await()`). The `finish([] { body; })` API waits for all tasks created in `body`, including transitively spawned tasks, before returning.

3.2 Enabling Resilience in HClib

We extended HClib to include the resilience constructs (Section 2.1), and the reference counting capability (Section 2.2).

As mentioned in section 2.1, to hold the `put` operations until equality checking succeeds, we need additional space within the `promise`. The normal `promise` can hold only one value that had been added to it using the `put` operation. For replication, however, all replicas perform the `put` operation and, therefore, we need N locations within the `promise` rather than one. To accommodate this, we extended the reference counting `promise` with an array to store N values so that we can perform majority voting among them. During a `put` operation inside a replication task, the i^{th} replica stores the value in the i^{th} location of the array. Similarly for replay or ABFT tasks, to hold the output inside the `promise` until it is published, we extended the reference counting `promise` to include a temporary storage. Unlike replication, which requires an array of temporary storage, a replay and ABFT `promise` needs only one temporary storage space since the replay happens sequentially.

We need to collect all the `put` operations within the resilient tasks so that they can be checked for errors after all replicas finish. For this purpose, we extended HClib with task-local storage. Each `put` operation in the replica with index zero (we assume all replicas perform the same `put` operations) adds the associated `promise` to the task-local storage. Finally, while merging the results

from the replicas, we fetch the promises from the task-local storage and check for equality on the data attached to those promises using an equivalence operator the user provides.

4 Evaluation

This section presents the results of an empirical evaluation of our runtime system, mostly on a single-node platform with a few experiments on a multi-node platform to show its viability in a distributed environment.

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. Cori uses Cray Aries interconnect with Dragonfly topology having a global peak bisection bandwidth is 45.0 TB/s. We used GCC 7.3.0 compiler for building the library and most benchmarks and Intel Compiler 18.0.1 for benchmarks that require MKL support.

Benchmarks: Our first benchmark is the stencil 1D benchmark that solves linear advection (a hyperbolic PDE). We implemented this using the Lax-Wendroff 3-point stencil. In this benchmark, we use 128 tiles of size 16,000 doubles, 128 time steps per iteration (each task advances its assigned tile 128 time steps), and 8,192 iterations. For our next benchmark, we solve heat diffusion (a parabolic PDE) on a 3D domain with periodic boundary conditions using a 7-point stencil. Here we use 16^3 cubes, each representing a subdomain of size 32^3, and run for 1,024 iterations. Our next benchmark is a tiled version of Conjugate Gradient (CG), which is an iterative method for solving sparse systems of linear equations. A square matrix from the "SuiteSparse" collection (52,804 rows/columns, 5,333,507 non-zeros) was set up with the CG method with 128 tiles and 500 iterations. Our fourth 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. We use two input strings of sizes 185,600 and 192,000, divided among 4,096 tiles arranged as 64×64. Our last benchmark is the Cholesky decomposition algorithm, which is used primarily to find the numerical solution of linear equations. Here we decompose a matrix of size 24,000×24,000 into tiles of size 400×400. We report the average of five runs for each experiment.

For the stencil benchmarks, we can detect corruption anywhere on a subdomain using physics-based checksums because conservation requires that the sum of values over the subdomain only changes by the flux through the subdomain boundary. For the Conjugate gradient and Smith-waterman benchmarks, there are not any sophisticated error detection mechanisms, so we simply return true, implying no error occurred. In the case when we want to inject faults, we pick a few instances of the error checking function to return false. Error-checking functions are expected to be domain/application dependent and are not the subject of this paper, and we emulate the scenario arising from a prescribed fault rate. The design of checksums for Cholesky decomposition is based on the work by Cao [6].

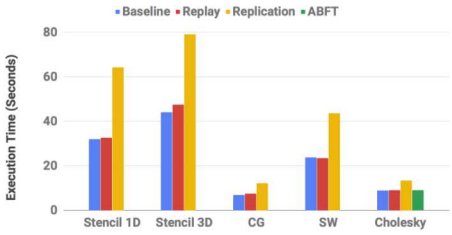

Fig. 1. Comparison of execution times of different resilience schemes on the five benchmarks without faults.

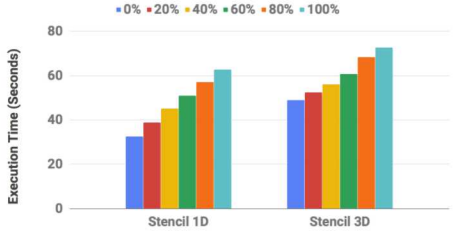

Fig. 2. Comparison of execution times of the stencil benchmarks while mixing replay and replication with percent of replication shown.

4.1 Performance numbers without failures

Single Resilience Technique To show the overhead of the resilient runtime, the execution time of the five benchmarks using various resilience techniques without failures is shown in Figure 1. For all the benchmarks, we used replay and replication to enable resilience. For the Cholesky benchmark, in addition to replay and replication, we included ABFT. From the figure, we can see that for the stencil benchmarks, some additional time is required for the replay variant compared to the baseline. For the stencil 1D benchmark, this accounts for less than 5% overhead whereas in stencil 3D the overhead is around 8%. A close examination reveals the overhead includes both the computation of the checksum and additional overhead from the replay runtime. For the Conjugate gradient benchmark, the replay runtime incurs an overhead of less than 10%. For the Smith-Waterman and Cholesky benchmarks, we did not notice any significant overhead while using replay. For the Cholesky benchmark, we also enabled ABFT and found that the time required for ABFT is comparable to that of replay.

When replication is used, we can see that the execution time increases for all the benchmarks. We expected the time to double because, in the absence of faults, duplication of the tasks occurs. However, for a few benchmarks, the execution time was significantly less than double primarily due to L3 cache reuse.

Mixing Resilience Techniques To illustrate that the various resilience techniques can be seamlessly combined, we also tried to mix replay and replication in the stencil benchmarks. On one end, the application only uses replay, and on the other end, it uses just replication. In between, the amount of replication is increased in increments of 20%. Figure 2 shows that the execution time increases linearly while mixing replay and replication. This implies that the increase directly corresponds to the additional cost for running replication and thus no additional overhead is involved.

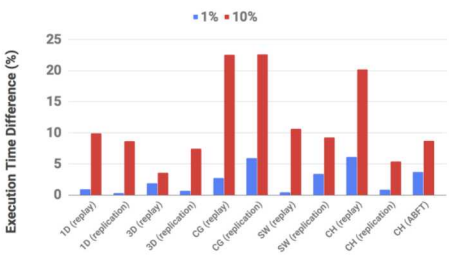

Fig. 3. Comparison of percentage of change in execution times w.r.t baseline of different resilience schemes with faults injected at 1% and 10% rate.

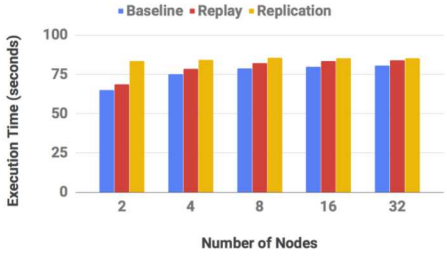

Fig. 4. Weak scaling of Stencil 1D with different number of nodes and resilience schemes with no fault injection

4.2 Performance numbers with failures

To check the effectiveness of our resilience mechanisms in the presence of soft errors, we ran all the benchmarks while introducing errors. We injected errors at a rate of 1% and 10%. Here, 10% implies that an error is injected into 10% of the total tasks. Figure 3 shows the execution time for various benchmarks and resilience techniques in the presence of faults. Here, also, we can see that the increase in execution time closely follows the amount of failure occurred. For the 10% failure rate, in most cases, the increase in execution time is also around 10%. Failures do not cause much time increase in case of ABFT because the ABFT error correction is very lightweight compared to others.

4.3 Performance numbers on multiple nodes

We run some preliminary experiments using stencil 1D with multiple nodes using MPI to measure the overhead of our implementation in multi-node environments[4]. We ran weak scaling of Stencil 1D by only increasing the number of tiles while keeping the same configuration of other parameters as the experiment on a single node. In Figure 4, we can see some performance degradation because of internode communication during a two-node run. However, the replication scheme worked well without degrading performance significantly because the communication incurred by MPI routines is overlapped with the replicated execution of tasks. For runs without resilience or with replay, MPI routines are called only when the original task or replayed task generates correct results, which causes a delay because it cannot be overlapped with other tasks. Thus, the execution time of such runs increases with more nodes.

[4] There is no resilience across nodes. We provide only single-node resilience and use MPI for communication. Resilience across nodes is part of future work.

5 Related work

Task Replication: Subasi *et al* [25] study a combination of task replication and checkpoint/restart for a task-parallel runtime, OmpSs [9]. Their checkpoint API is integrated with the input data parameters of OmpSs directives to protect the task input. They also suggested deferring launch of the third replica until duplicated tasks report a failure. However, the mixing with other resilience techniques and analysis of the performance penalties are yet to be studied.
Task Replay: Subasi *et al* [24] also study a combination of task replay and checkpoint/restart for OmpSs. As with task replication, checkpoint/restart is utilized for preserving the input of tasks. During the execution of a task, errors notified by the operating system trigger a replay of the task using the input data stored in the checkpoint. Cao *et al* [6] has a similar replay model. However, the drawback of these approaches is a lack of mitigation for failure propagation, as they assume *reliable* failure detection support, e.g., by the operating system, which is not always available. Our approach provides a general interface that allows user-level failure detection.
ABFT: Cao *et al* [6] also discuss an algorithm-based fault tolerance for tiled Cholesky factorization [16] in the PaRSEC runtime. However, they do not discuss their user-visible APIs in terms of general applicability, while our approach provides a general support for ABFT.

6 Conclusions and Future Work

The traditional checkpoint/restart (C/R) approach for resilience was designed to support the bulk-synchronous MPI programming model under the assumption that failure is a rare event. However, C/R is not well suited for supporting higher-frequency soft errors or unexpected performance anomalies. The resilient-AMT idea for applications mitigates the shortcoming of traditional C/R, so as to support scalable failure mitigation. Task decomposition allows localization and isolation of failures in the resilient-AMT framework, and thus keeps the recovery inexpensive. Our work realizes the four resilience programming concepts suggested by Heroux [14]. Task boundary helps to perform **Local Failure Local Recovery** for scalable application recovery. The task replication and replay APIs allow **selective reliability**; the use of replication and replay on individual tasks can be at the user's discretion. The task replay and ABFT APIs enable **skeptical programming**, which can incorporate inexpensive user-defined error detection. The response to an error is either task replay (rollback) or recovery (application-specific correction). The AMT execution model **relaxes the assumption of bulk-synchrony** of conventional parallel programs.

In the future, we would like to extend our resilient-AMT approach to support both intra-node and inter-node resilience (MPI based communication). Another direction is to combine both replication and replay mechanisms in an "eager replay" approach. During eager replay, if extra resources are available, the replay task can run multiple copies instead of waiting for the task to finish and select

the correct output from the replicas using a `selection` function. Our current approach also depends on the use of a user-provided `equals` function to check for equivalence of data which could be automated using a compiler. Although we support the use of nested non-resilient tasks within a resilient task, nesting of resilient tasks is a topic of future research. Also the restriction of side-effect free tasks can be relaxed by using idempotent regions as task boundaries [19]. Our current approach only supports one level of checkpointing, with access to checkpoints of parent tasks. If that execution fails again, we may need to recover from checkpoints of further ancestors (multi-level checkpointing), as part of our future work. Another direction is to study the characteristics of faults [18, 23] and perform fault injection [10, 12] to efficiently and extensively cover them.

References

1. HClib Resilience Branch. `https://github.com/srirajpaul/hclib/tree/feature/resilience`, accessed: 2019-06-14
2. Augonnet, C., et al.: StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. Concurrency and Computation: Practice and Experience **23**(2), 187–198 (2011). https://doi.org/10.1002/cpe.1631
3. Bauer, M., et al.: Legion: Expressing locality and independence with logical regions. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 66:1–66:11. SC '12 (2012). https://doi.org/10.1109/SC.2012.71
4. Bennett, J., et al.: ASC ATDM Level 2 Milestone #5325: Asynchronous Many-Task Runtime System Analysis and Assessment for Next Generation Platform. Tech. Rep. SAND2015-8312, Sandia National Laboratories (September 2015)
5. Bosilca, G., et al.: PaRSEC: Exploiting Heterogeneity to Enhance Scalability. Computing in Science Engineering **15**(6), 36–45 (Nov 2013). https://doi.org/10.1109/MCSE.2013.98
6. Cao, C., et al.: Design for a soft error resilient dynamic task-based runtime. In: 2015 IEEE International Parallel and Distributed Processing Symposium. pp. 765–774 (May 2015). https://doi.org/10.1109/IPDPS.2015.81
7. Cappello, F., et al.: Toward exascale resilience: 2014 update. Supercomput. Front. Innov.: Int. J. **1**(1), 5–28 (Apr 2014). https://doi.org/10.14529/jsfi140101
8. Chung, J., 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. pp. 58:1–58:11. SC '12 (2012). https://doi.org/10.1109/SC.2012.36
9. Fernández, A., 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. pp. 601–612 (2014). https://doi.org/10.1007/978-3-319-14313-2_51
10. Georgakoudis, G., et al.: Refine: Realistic fault injection via compiler-based instrumentation for accuracy, portability and speed. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 29:1–29:14. SC '17 (2017). https://doi.org/10.1145/3126908.3126972
11. Grossman, M., et al.: A pluggable framework for composable hpc scheduling libraries. In: 2017 IEEE International Parallel and Distributed Pro-

cessing Symposium Workshops (IPDPSW). pp. 723–732 (May 2017). https://doi.org/10.1109/IPDPSW.2017.13
12. Guan, Q., et al.: F-sefi: A fine-grained soft error fault injection tool for profiling application vulnerability. In: 2014 IEEE 28th International Parallel and Distributed Processing Symposium. pp. 1245–1254 (2014). https://doi.org/10.1109/IPDPS.2014.128
13. Hayashi, A., et al.: Chapel-on-x: Exploring tasking runtimes for pgas languages. pp. 5:1–5:8. ESPM2'17, ACM, New York, NY, USA (2017). https://doi.org/10.1145/3152041.3152086
14. Heroux, M.A.: Toward Resilient Algorithms and Applications. http://arxiv.org/abs/1402.3809 (2014)
15. Huang, K.H., Abraham, J.A.: Algorithm-based fault tolerance for matrix operations. IEEE Transactions on Computers **C-33**(6), 518–528 (June 1984). https://doi.org/10.1109/TC.1984.1676475
16. Jeannot, E.: Performance analysis and optimization of the tiled cholesky factorization on numa machines. In: Proceedings of the 2012 Fifth International Symposium on Parallel Architectures, Algorithms and Programming. pp. 210–217. PAAP '12 (2012). https://doi.org/10.1109/PAAP.2012.38
17. Kaiser, H., et al.: Parallex an advanced parallel execution model for scaling-impaired applications. In: 2009 International Conference on Parallel Processing Workshops. pp. 394–401 (2009). https://doi.org/10.1109/ICPPW.2009.14
18. Li, D., et al.: Classifying soft error vulnerabilities in extreme-scale scientific applications using a binary instrumentation tool. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 57:1–57:11. SC '12 (2012). https://doi.org/10.1109/SC.2012.29
19. Liu, Q., et al.: Compiler-directed lightweight checkpointing for fine-grained guaranteed soft error recovery. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 20:1–20:12. SC '16 (2016). https://doi.org/10.1109/SC.2016.19
20. Mattson, T.G., et al.: The open community runtime: A runtime system for extreme scale computing. In: 2016 IEEE High Performance Extreme Computing Conference (HPEC). pp. 1–7 (Sept 2016). https://doi.org/10.1109/HPEC.2016.7761580
21. Moody, A., 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. pp. 1–11. SC '10 (2010). https://doi.org/10.1109/SC.2010.18
22. Plank, J.S., Li, K., Puening, M.A.: Diskless checkpointing. IEEE Trans. Parallel Distrib. Syst. **9**(10), 972–986 (Oct 1998). https://doi.org/10.1109/71.730527
23. Shantharam, M., et al.: Characterizing the impact of soft errors on iterative methods in scientific computing. In: Proceedings of the International Conference on Supercomputing. pp. 152–161. ICS '11 (2011). https://doi.org/10.1145/1995896.1995922
24. Subasi, O., et al.: 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. https://doi.org/10.1109/PDP.2015.17
25. Subasi, O., 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). pp. 452–457 (May 2017). https://doi.org/10.1109/CCGRID.2017.40