Received ; | Revised ; | Accepted
DOI: xxx/xxxx

Node Failure Resiliency for Uintah Without Checkpointing

Damodar Sahasrabudhe* | **Martin Berzins** | **John Schmidt**

[1]Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, Utah, USA

Correspondence
*Damodar Sahasrabudhe
Email: damodars@sci.utah.edu

Present Address
Scientific Computing and Imaging Institute
University of Utah
72 Central Campus Dr
Salt Lake City, UT 84112

Summary

The frequency of failures in upcoming exascale supercomputers may well be greater than at present due to many-core architectures if component failure rates remain unchanged. This potential increase in failure frequency coupled with I/O challenges at exascale may prove problematic for current resiliency approaches such as checkpoint restarting, although the use of fast intermediate memory may help. Algorithm-Based Fault Tolerance (ABFT) using Adaptive Mesh Refinement (AMR) is one resiliency approach used to address these challenges. For adaptive mesh codes, a coarse mesh version of the solution may be used to restore the fine mesh solution. This paper addresses the implementation of the ABFT approach within the Uintah software framework: both at a software level within Uintah and in the data reconstruction method used for the recovery of lost data. This method has two problems: inaccuracies introduced during the reconstruction propagate forward in time, and the physical consistency of variables such as positivity or boundedness may be violated during interpolation. These challenges can be addressed by the combination of two techniques: 1. a fault-tolerant MPI implementation to recover from runtime node failures, and 2. high-order interpolation schemes to preserve the physical solution and reconstruct lost data. The approach considered here uses a "Limited Essentially Non-Oscillatory" (LENO) scheme along with AMR to rebuild the lost data without checkpointing using Uintah. Experiments were carried out using a fault-tolerant MPI - ULFM to recover from runtime failure, and LENO to recover data on patches belonging to failed ranks, while the simulation was continued to the end. Results show that this ABFT approach is up to 10x faster than the traditional checkpointing method. The new interpolation approach is more accurate than linear interpolation and not subject to the overshoots found in other interpolation methods.

KEYWORDS:
Resilience, MPI fault tolerance, interpolation, checkpointing, parallel computing, Uintah

1 | INTRODUCTION

The move to a new generation of supercomputers with peak performance at exascale over the next five years or so presents significant challenges. One of the options to reach exascale within the expected power budget is to increase the number of cores in present leading architectures from 10 million to 100 million. Chips are expected to contain 100x more processing elements than those currently used, which may increase the processing components from approximately 500,000 in today's systems to a substantial fraction of one billion by 202X (1). At present, exascale systems running millions of cores are expected to experience numerous faults every day. This potential challenge has spurred research in the area of resiliency. For example, (2, 3) describe some approaches to resiliency based on the types of problems that occur. These problems may be categorized as soft failures due to bit flips or hard failures due to core, node or communications failures. At the same time, it is not clear whether research in computer architecture can reduce the

[0]**Abbreviations:** AMR, Adaptive Mesh Refinement; LENO, limited ENO; ULFM, User Level Failure Mitigation

number of failures. In any case, with the potential for increased failure due to faults, a development path is needed to handle any potential resilience issues. The exascale committee report predicts that the time required for an automatic or the application-level checkpoint/restart will exceed the mean time to failure (MTTF) of a full system (1). As a result, the traditional recovery technique of checkpointing and recomputing from the last checkpoint may prove problematic in the development of exascale computing in the near future.

This challenge makes Algorithm-Based Fault Tolerance (ABFT) a more attractive option. With ABFT, data lost during failure can be reconstructed at runtime, because programmers are aware of the algorithms and functionality of the particular problem being solved. Since recovery occurs at runtime without using checkpointing (and hence without disk access), ABFT has the potential to be faster than traditional checkpoint/recovery.

Uintah (4, 5) is an open-source asynchronous many-task software framework. It can be used to generate solutions of partial differential equations (PDEs), and to model complex multi-physics and multi-scale problems

in three space dimensions and a time dimension. Uintah has demonstrated excellent strong scaling up to 786K cores on supercomputers, including NSF Stampede, DOE Titan and DOE MIRA (6, 7, 8). Applications built using Uintah have been able to scale on CPUs, GPUs and Intel's Knights Landing architectures (9). Uintah's task-based approach allows users to write simulation components to solve a particular problem and hides communications, scheduling, load balancing, etc. from users by executing tasks through a runtime system. Based on the earlier observations, a different resilience mechanism might be needed to run Uintah at exascale. This work intends to explore an ABFT resiliency approach for Uintah by addressing the future work challenge posed by Dubey (10): using lower fidelity solutions on surviving compute nodes to rebuild a solution for failed nodes.

Uintah supports Adaptive Mesh Refinement (AMR) and provides built-in tasks for coarsening/refining patches as required by ABFT. In introducing resiliency to Uintah, this work makes the following **contributions:**

- An ABFT solution for Uintah is constructed in which:
  - Node failures are detected using User-Level Failure Mitigation (ULFM)(11). ULFM is a modified version of OpenMPI that supports failure detection.
  - Surviving ranks are brought back to a stable state, and tasks from lost ranks are redistributed to surviving ranks.
  - Lost patches are recovered using interpolation, and the normal execution of Uintah is continued.
- This ABFT solution is shown to perform faster than the traditional checkpointing method.
- An accurate physics-constrained interpolation for recovery is used to reconstruct the solution. Evidence going back to (12) suggests that for problems involving integration forward in time, new values should be calculated with sufficient accuracy such that interpolation errors do not pollute the remainder of the time integration. The approach taken here is to extend Dubey's work on recovering from node failure by using a simple form of the limited ENO interpolation (LENO) scheme suggested by Berzins (13) that preserves the positivity and boundedness of the solution. This preservation of the physical bounds on the solution values is important in many real-life engineering applications, such as combustion (14) and weather forecasting (15).
- A novel advection-reaction type problem is developed and used in one- and three-dimensional cases to show the importance of using interpolation that respects physical bounds on the solution. A three-dimensional version of Burgers' equation is used to show the need for high-accuracy interpolation.

2 | RELATED WORK

Research has increasingly focused on the area of resiliency to meet the exascale challenge. Faults can be handled at a system level through either Hardware-Based Fault Tolerance (HBFT) or through Systems software-Based Fault Tolerance (SBFT) independent of applications (16).

Early works regarding fault tolerance, including dynamic MPI programs with checkpointing and resilient versions of MPI, are described in (17, 18) with a relatively recent summary found in (19, 20). Cappello (21) has summarized recent developments in resiliency that targets exascale. Some key checkpointing approaches include BLCR checkpointing (22), adaptive checkpointing (23), hybrid checkpointing (24), data aggregation (25), and incremental checkpointing (26). Hussain (27) carried out a theoretical study explaining how non-identical failure distribution among nodes can help in achieving different levels of replication and deliver faster performance.

Checkpointing and restarting are widely used but are inherently time-consuming. Dauwe (2) analyzes different resiliency techniques such as rollback recovery with checkpointing and restarting, or non-blocking multiple levels of checkpointing (28), message logging and full or partial redundancy, along with their performance characteristics.

2.1 | Algorithm-Based Fault Tolerance

Algorithm-Based Fault Tolerance (ABFT) (16, 29) uses application-specific algorithms to construct more efficient resilience techniques. Several ABFT techniques have been developed for different algorithms, e.g., ABFT for matrix-matrix multiplications-based programs (30, 31), which have been further extended to linear algebra routines/benchmarks (32, 33). The GVR library (34, 35) can maintain and present a global view of distributed arrays along with versioning of these arrays. GVR provides programmers with the capability to implement ABFT by accessing remote coarse mesh data structures. Depending on the algorithm and the level of its failure, an appropriate version of an array can be retrieved from GVR to implement the desired recovery routine (29, 36). Dubey handles soft failures such as errors due to bit flips at different levels, such as cell or box level, using ABFT (10, 36). She describes two approaches to handle hard failures such as core or node failure - one based on ABFT and the other on retrieving missing data from GVR - but she chooses the latter (10, 36). The solution adopted here is the less expensive but more complex approach that Dubey (10) described in which the solution is reconstructed on the missing mesh patches due to a node failure. The accuracy of the data must be ensured while rebuilding the solution on fine mesh patches using coarse data values. Such issues arise even in standard AMR calculations (12) but are not often referenced. Rebuilding fine mesh data using coarse mesh data may result in reduced accuracy on the fine mesh patch that persists as the solution evolves. The use of high-order interpolation schemes to regenerate this data helps address the accuracy issue. These schemes, however, may violate important physical properties of the solution, such as the need for positivity, and may introduce spurious maxima "overshoots" or spurious minima "undershoots". For this reason, it is important to consider using interpolation approaches that respect any underlying desired physical bounds on the solution.

2.2 | Fault Detection Mechanisms for a Node Failure

The ABFT approach is feasible only if a reliable fault detection mechanism exists. Many research codes are available, but, to the best of the authors' knowledge, only two proposed/experimental standards are in place:

- Fault Tolerant MPI (FT-MPI) (37)
- User Level Failure Mitigation (ULFM) (11)

Both FT MPI and ULFM provide user-level APIs that can be called from a custom error handler. These APIs can be used to detect failed ranks and form a consistent view of the MPI world across all surviving ranks. Both approaches can detect and possibly recover up to n-1 failed processes out of n processes. In both standards, the responsibility for recovery rests with application developers.

Many studies over the last several years have attempted to address resilience. An early example was the novel architecture of Starfish (17) that was based on combining group communications technology and checkpointing. Varma (38) developed a protocol to detect node failures and create a consistent view of active nodes among MPI ranks. Morris et al. (39) implemented a fault-tolerant ULFM and ABFT-based PDE solver. However, they used a server-client model in which servers, immune to failure, maintain the "state" of the system and hand over computational tasks to clients, which are allowed to fail. This design ensures that a client node failure hampers only data and not the state, but restricts failure to only parts of the algorithm that are executed on the client nodes. Uintah, on the other hand, does not use a server-client model. Each rank maintains its own state and can fail at any given time. Hence, a resilient version of Uintah needs to recreate tasks and rebuild patches belonging to "lost" ranks.

3 | UINTAH FRAMEWORK

Uintah (4, 5) is an open-source asynchronous massively parallel many-task software framework. It can be used to generate solutions of partial differential equations (PDEs) and to model complex multi-physics and multi-scale problems. Uintah has demonstrated excellent strong scaling up to 786K cores on supercomputers, including NSF Stampede, DOE Titan and DOE MIRA (6, 7, 8). Applications using Uintah have been able to scale on CPUs, GPUs, and Intel's Knights Landing architectures (9). Uintah allows users to write simulation components in the form of multiple tasks with data dependencies. Uintah compiles tasks into a directed acyclic graph (DAG) with edges serving as dependencies. Tasks are executed as dependencies are satisfied, which keeps communications, scheduling, load balancing, etc. hidden from users.

Uintah computes solutions of PDEs with a variety of timestepping methods, such as the Forward Euler method. For every vertex of every cell, Uintah calculates the solution at the next timestep:

$$u(t+\Delta t) = u(t) + \Delta t.F(u(t))$$

where $u(t+\Delta t)$ is the value after time Δt and $F(u(t))$ defines the time derivative of $u(t)$. More information about Uintah can be found in references (4, 5, 7).

The most important feature of Uintah for this work is the support for Adaptive Mesh Refinement (AMR), as described in detail in (40). Uintah employs a grid of cubic mesh cells grouped into patches. The grid can be divided into "levels" of patches. With each level, patches can be refined to a higher resolution or can be coarsened back to a lower resolution. The refinement ratio can be specified for each level to make the transition from fine to coarse patches (or vice versa). Uintah provides the capability to use a fine mesh locally on a node and replicate coarse patches on other nodes to reduce the communication overhead (7).

4 | IMPLEMENTATION OF RESILIENCE IN UINTAH

This implementation of resilience in Uintah introduces a systematic way to detect and handle failures, as shown in Figure 1 and Algorithm 1. Statements highlighted in Algorithm 1 indicate the changes that were made to achieve resiliency. The algorithm also provides a short description of existing steps for task graph compilation, dependency creation, automated MPI message generation, load balancing, task scheduling, execution, etc. More details on the existing task execution model of Uintah can be found in references (4, 5, 7). ULFM was chosen as the resilient MPI implemen-

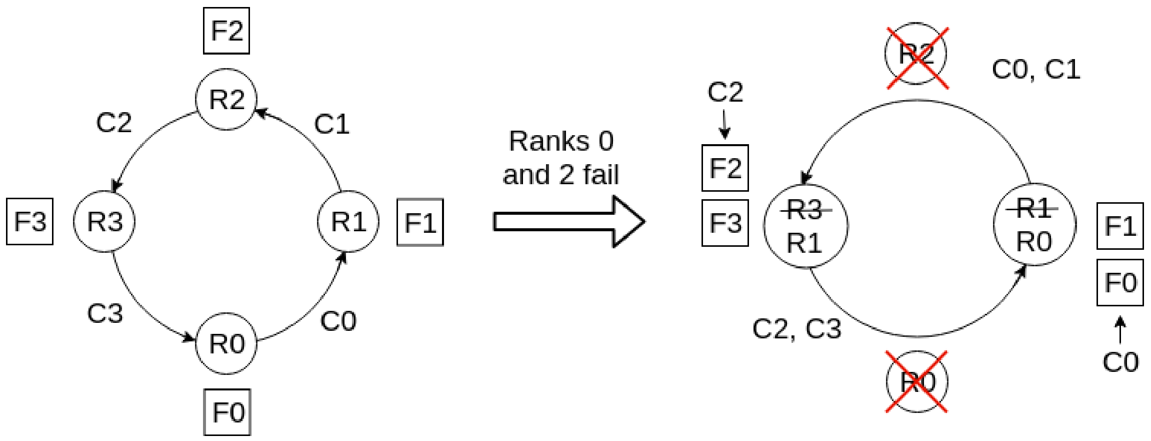


FIGURE 1: Node failures and patch recovery for MPI ranks R0, R1, R2 and R3, with fine patches F0, F1, F2 and F3 and coarse patches C0, C1, C2 and C3.

tation because of two important features: ULFM provides an abstraction to "revoke" MPI calls, and it can be used by surviving processes to revoke any pending calls made to the failed rank. Revoking pending MPI calls avoids the deadlock of processes that are waiting on the failed rank. ULFM also provides another abstraction to "shrink" the global communication handle. This abstraction removes failed ranks from the set of active ranks and returns a new global communication handle containing surviving processes that can be used for recovery and post-crash execution.

The following steps illustrate the resiliency process:

- **Patch Coarsening**: A new task to generate coarse patches is created and scheduled. This task utilizes the existing AMR capabilities of Uintah with a refinement ratio of 1:2. Thus, for every eight cells of a fine patch in three dimensions, there is one cell on a coarse patch. The refinement ratio can be changed depending on accuracy and performance requirements; however, both coarsening and interpolation tasks should use the same ratio. At the moment, coarsening is done by simply discarding alternate values in all three dimensions. For the coarsening

Algorithm 1 Resilient Uintah Algorithm:

```
 1: Initialize MPI world. Set error handler to static ErrorHandler method in new class Resiliency.
 2: Read input file (.ups file) to get the problem specification.
 3: Create instances of the Scheduler, Load Balancer, Data Archiver and execution component depending on input file and call the Simulation
    Controller.
 4: Create an instance of Resiliency class, and save pointers to Processor Group (which holds information about MPI world), Simulation
    Controller, Scheduler, Load Balancer, Data Archiver and execution component to the Resiliency instance.
 5: Create new tasks: 1. to coarsen patches with a dependency on timeAdvance task. 2. A dummy task with a dependency on coarsen task.
 6: Call "schedule initialize" and "schedule timestep" methods of the components provided by users of Uintah. These methods create and add tasks
    to the Scheduler instance.
 7: Create Task Graphs (DAG) for all the tasks and their dependencies.
 8: Determine neighborhood processors.
 9: Add extra tasks to send data to dependent processes. Create extra tasks for neighboring processors. These tasks are not executed on the current
    processor but are used for creating dependencies.
10: Assign processing resources for each task.
11: Call EXECUTE()
12: Finalize MPI.
13:
14: procedure EXECUTE
15:     for i = 1 to number of timesteps do
16:         Advance Data Warehouse from previous timestep to current timestep.
17:         for every task in task graph do
18:             Post MPI Receive messages for dependencies.
19:             Execute the task (i.e., the function pointer provided by the users in Scheduler timestep function)
20:             Post MPI Send messages for dependencies.
21:         end for
22:     end for
23: end procedure
24:
25: procedure ERRORHANDLER
26:     Use ULFM APIs MPIX_Comm_failure_ack, MPIX_Comm_agree, and MPIX_Comm_failure_get_acked to find out failed ranks, and create a
    consistent picture of failed ranks globally.
27:     Call ULFM API MPIX_Comm_shrink to get new global MPI communicator handle, which excludes failed ranks.
28:     Set newRank = oldRank
29:     for every failed rank do
30:         newRank = newRank > failedRank ? newRank-1 : newRank
31:     end for
32:     Call MPI_Comm_split to assign ranks as per newRank.
33:     Update comm instances stored in class ProcessorGroup and add error handler for new communicator.
34:     Clear existing communication queues.
35:     Reassign 'execution rank' to patches using same logic of newRank.
36:     for every task graph do
37:         Call load balancer method to recreate neighborhood, because updating ranks and 'patch execution ranks' changes neighbors.
38:         Reassign task resources as per patches.
39:         Clear dependencies of every task within a task graph.
40:     end for
41:     Merge any reductions/output tasks.
42:     Create/update Send Old Data tasks to accommodate new neighbors.
43:     Create new tasks corresponding to neighbors on every rank, which is necessary to create dependencies used in MPI communication.
44:     Allocate memory for failed patches.
```

Algorithm 1 Resilient Uintah Algorithm: (continued)

```
45:     for every task graph do
46:         Create dependencies.
47:         Assign MPI message tags.
48:         Recompute local tasks.
49:     end for
50:     Schedule and execute an interpolation task for failed patches using sub-scheduler.
51:     Call EXECUTE(): This continues execution of timesteps.
52: end procedure
```

task a “required” dependency is added from the “timeAdvance” task, which is the main task used to implement the timestepping algorithm within Uintah. Because of task dependencies, Uintah schedules the coarsening task after timeAdvance and feeds the fine patches computed by timeAdvance as input to the coarsening task. Figure 1 shows an MPI world with four ranks. Ranks R0 through R3 compute fine patches F0 through F3 and then compute coarse patches C0 through C3, respectively.

- **Coarse Patch Exchange**: To exchange coarse patches, each rank creates an empty dummy task and adds a dependency on a coarse patch computed by the $(n-1)^{th}$ rank. The dummy task does not compute anything, but because of the dependency on the $(n-1)^{th}$ rank, Uintah generates MPI messages to exchange patches. Thus, each process with rank n sends its coarse patches to the $(n+1)^{th}$ rank and receives coarse patches from the $(n-1)^{th}$ rank in a circular fashion, as shown in Figure 1. This logic has several aspects: if two consecutive ranks fail, then both the fine and the coarse mesh patches are lost, causing an irreversible loss; or all tasks of failed rank get assigned to one neighboring rank, creating a severe load imbalance. However, the patch exchange logic is flexible and can be easily updated to send coarse patches to more than one rank. In the future, better load balancing can be achieved by using Uintah’s dynamic load balancer to distribute orphan tasks evenly across survivor ranks. Coarsening and exchange tasks can also be scheduled at a fixed interval of timesteps rather than scheduling them for every timestep.

- **Determine the Faulty Nodes / Ranks**: MPI rank failure can be detected by changing the default MPI error handler from MPI_ERRORS_ARE_FATAL to a custom error handler in the new "Resiliency" class. This "Resiliency" class is instantiated after instantiating all other infrastructure classes and saves pointers to instances of Scheduler, Load Balancer, Simulation Controller, Application State and MPI World. The error handler can then access these instance pointers to get the latest “state” of the simulation, and the framework can restore the context after an exception is caught.

  ULFM APIs (MPIX_Comm_failure_ack, MPIX_Comm_agree and MPIX_Comm_failure_get_acked) are used to determine failed ranks, create a uniform picture of failed ranks across all surviving ranks, create the new global communicator excluding failed ranks and invoke recovery routines.

- **Reassignment of ranks and patches**: When the n^{th} rank fails, the $(n+1)^{th}$ rank takes over patches of the n^{th} rank, and the rank of the process is also updated from $(n+1)$ to n. Subsequently, each rank greater than the failed rank is decremented by one. When multiple ranks fail, the same logic is iterated for every failed rank. The patch to rank assignment also uses the same logic. This process is shown in Figure 1 where ranks R0 and R2 fail. Rank R1 then becomes the new rank R0 and rank R3 becomes the new rank R1. The orphan fine patches F0 and F2 are now allocated to ranks R0 and R1 (old ranks R1 and R3). When the ranks are updated, the patch to rank assignment for patches F1 and F3 is also updated to R0 and R1, respectively. Algorithm 1 calls MPIX_Comm_split to ensure the rank are reassigned following the same reassignment strategy.

- **Updating the task graph**: The recovery process creates new tasks and/or changes the ownership of existing tasks, except for the reduction tasks that belonged to failed ranks. The reduction tasks are skipped to avoid additional calls to MPI_Reduce.

  New send data tasks are created on behalf of new neighbors to ensure correct dependency creation. All existing dependencies are then deleted and recreated to reflect new tasks and ranks for the automated MPI message generation.

- **Data Recovery**: Finally, the error handler reallocates memory for the adopted fine patches and schedules an interpolation task to generate fine patches from coarse patches received earlier. A sub-scheduler is used to schedule the interpolation task, because it is a one-time task and is not repeated for subsequent timesteps. The sub-scheduler creates a new instance of the task graph containing only interpolation tasks and avoids modifying the main task graph, which gets executed every timestep. Any MPI communication needed for halo exchange is automatically handled by the sub-scheduler utilizing the Uintah infrastructure. As shown in Figure 1, new ranks R0 and R1 rebuild fine patches F0 and F2 using coarse patches C0 and C2. Uintah provides three types of variables - Node Centered (with data points at eight vertices of a cubic cell), Cell Centered (with data points at the center of a cubic cell) and Face Centered (with data points at centers of six faces of a cubic cell). In three space dimensions, the algorithm uses Tensor Product interpolation, which is a standard approach described in (41). The application of this idea to the Newton polynomial approach described in the next section is an adaptive form of the algorithm that dates back to Narumi (42). The challenge of addressing cell-vertex and cell-centered meshes requires a complex but straightforward application of the underlying Tensor Product approach. At each stage of this Tensor Product interpolation, boundedness in the solution is enforced for all intermediate values. The general interpolation approach is similar to the way that coarse-fine interfaces are treated in many adaptive mesh codes (43, 44).

- **Continued Execution**: Once the failed patches are recovered, execution continues in the usual fashion with the revised (reduced) MPI world. Figure 1 shows that ranks R0 and R1 will continue executing tasks on patches F0, F1 and F2, F3, respectively. Rank R0 will also have coarse patches C2 and C3, and rank R1 will have coarse patches C0 and C1.

  The disadvantage of using the reduced communicator instead of spawning new ranks is the increase in the execution time for subsequent timesteps due to fewer MPI processes and nodes.

  Recreating instances of all components in newly spawned ranks and bringing those instances to the state of the current timestep is more involved than using the surviving ranks to take over patches with the existing state. This approach can be considered in the future.

The existing design keeps recovering from failures until only one rank remains at the end provided that certain conditions hold. This requires that the maximum number of *simultaneous* failures is limited to half the ranks. In addition, the pattern of patch exchange dictates that two neighboring ranks should not fail at the same time; otherwise the system cannot recover from a failure. While failures (which may or may not be concurrent) can occur one after another until only one rank remains, e.g., in Figure 1, first failure ranks (ranks R0 and R2) and recovery ranks (remaining ranks R1 and R3 now will be renamed ranks R0 and R1) end up owning patches R0:F0, F1, and R1:F2, F3, respectively. If rank R0 fails now, R1 will be renamed as R0, and R0 will take over all the patches F0 through F3.

5 | INTERPOLATION METHODS

One of the important factors in choosing an interpolation method is its accuracy. In real-life numerical simulations, it is essential to maintain physical properties such as positivity of the solution and to avoid the introduction of any spurious overshoots or undershoots. Weather forecasting (15) and combustion problems (14) are two such problems in which non-physical solution values may not match with the underlying physics. Previous work (12) demonstrated a need for more accurate methods than linear interpolation and suggested that cubic interpolation is important if the underlying discretization method is first or second order.

The difficulty with using cubic interpolation is possible overshoots and undershoots, i.e., interpolated values going above or below the limit of physically possible values. These overshoots and undershoots can cause errors by violating the given set of laws governing the system. For example, chemical concentrations cannot be negative. ENO methods and their extensions such as WENO (45, 46) provide a polynomial interpolant that preserves positivity in many but not all cases.

A simple approach inspired by Limited ENO (LENO) (13) is used to curtail overshoots and undershoots: allow only those divided differences that are sufficiently small. Alternatively, some of the other approaches considered in (13) might be adopted, such as the use of cubic splines.

The starting point is to use ENO interpolation schemes (13, 45) that employ the Newton divided difference form of the interpolating polynomial. For example, the cubic interpolation between known data points $U[x_i]$ and $U[x_{i+1}]$ to calculate $U[x]$ is given by

$$U[x] = U[x_i] + (x - x_i)U[x_i, x_{i+1}] + T3 + T4$$

where $U[x_i, x_{i+1}] = \frac{U[x_{i+1}] - U[x_i]}{x_{i+1} - x_i}$ and the terms $T3$ and $T4$ are products of a "multiplier" π and a "divided difference" δ: $T = \pi.\delta$. More terms may be added to obtain still higher order interpolation. The recursive formula for calculating the multiplier and divided difference for subsequent terms is $\pi_k = (x - x_i)(x - x_{i+1})...(x - x_{i+k-1})$ and $\delta_k = U[x_i, x_{i+1}, ..., x_{i+k}]$ where

$$\delta_k = \frac{U[x_{i+1}, x_{i+2}, ..., x_{i+k}] - U[x_i, x_{i+1}, ..., x_{i+k-1}]}{x_{i+1} - x_i}$$

The third term ($T3$) can be calculated using either $U[x_{i-1}]$ or $U[x_{i+2}]$, and δ will depend on what the next interpolation points are chosen to be. Using x_{i-1} gives δ as

$$\delta = U[x_{i-1}, x_i, x_{i+1}] = \frac{U[x_i, x_{i+1}] - U[x_{i-1}, x_i]}{x_{i+1} - x_{i-1}}$$

and using x_{i+2} for δ gives

$$\delta = U[x_i, x_{i+1}, x_{i+2}] = \frac{U[x_{i+1}, x_{i+2}] - U[x_i, x_{i+1}]}{x_{i+2} - x_i}$$

The ENO method (46) chooses the value of delta with the smallest absolute value, i.e., $min(|U[x_{i-1}, x_i, x_{i+1}]|, |U[x_i, x_{i+1}, x_{i+2}]|)$ to calculate T3. In the same way, subsequent terms can be computed by first calculating δ for points on either side, and picking the δ with the smallest absolute value. π is calculated using terms picked up in the previous δ. ENO methods work well for many problems, but they may prove troublesome for problems bounded by physics (or any domain-specific) constraints. Berzins (13) used an adaptive ENO method that imposes conditions on the ratios of divided differences and limited the polynomial order if those conditions are violated. A simpler approach used here is to calculate the linear, quadratic and cubic interpolation terms for the desired point, and then to pick the highest order value that satisfies the physical requirements of the solution. These requirements may include that the interpolated values are positive and bounded above, or that the interpolated values lie between the two nearest coarse mesh values (13). In the worst case, linear interpolation is used. For example, when using an even mesh in one space dimension, the linear, left quadratic and cubic interpolants that approximate the solution values at $x_{i+1/2}$, as denoted by $u^{linear}_{i+1/2}, u^{quadratic}_{i+1/2}, u^{cubic}_{i+1/2}$, are then given by

$$u^{linear}_{i+1/2} = \frac{1}{2}(u_i + u_{i+1}) \tag{1}$$

$$u^{quadratic}_{i+1/2} = \frac{1}{8}(-u_{i-1} + 6u_i + 3u_{i+1}) \tag{2}$$

$$u^{cubic}_{i+1/2} = \frac{1}{16}(-u_{i-1} + 9u_i + 9u_{i+1} - u_{i+2}) \tag{3}$$

It is straightforward to see that if the condition $u_{i-1} + u_{i+2} > 9(u_i + u_{i+1})$ holds, then $u^{cubic}_{i+1/2}$ will be negative, and either the quadratic or linear values should be used. For the quadratic case, if $u_{i-1} > 6u_i + 3u_{i+1}$, then the linear polynomial must be used. In effect, this approach employs a subset of the possible ENO interpolants while still ensuring that the interpolant is bounded. In the experiments that follow, the accuracy of linear, cubic ENO and Limited ENO interpolation methods will be shown. The extension to three space dimensions is achieved using a standard tensor product approach described in the previous section.

6 | EXPERIMENTS AND RESULTS

The computational experiments described below in Sections 6.1 and 6.2 were designed to compare the accuracies of three interpolation methods, linear, cubic ENO and cubic LENO, for reconstructing fine patches from coarse patches. Section 6.3 describes further experiments that compare the performance of the ABFT approach (Algorithm 1) against the performance of checkpointing/restarts.

6.1 | A 1D Advection-Reaction Test Problem

A simple but challenging test problem that illustrates some of the challenges with the interpolation methods described above is given by the advection reaction equation

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} = 40(c-1)u(1-u), \quad where\ (x,t) = [0,2]x(0,1.5]. \tag{4}$$

The left boundary condition and the initial condition are given by

$$u(x,t) = 0.5(1 - tanh(20(x - ct) - 4.0)) \tag{5}$$

The discretization method used is the explicit flux limited scheme with forward Euler integration (47). The source term in this problem is similar to that in (14, 47). Solution values outside the range $[0,1]$ will cause the source term to change sign, with potentially catastrophic results, particularly if the parameter c is not close to one. At every ten timesteps, failure is modeled by the fine mesh solution at every other fine mesh point being replaced by an interpolated value from the coarse mesh. This model represents every fine mesh patch failing at every ten timesteps and is a particularly stringent test of the need for appropriate spatial interpolation routines. In all cases, the error is the difference between the computed solution and the exact solution.

c	1.0	0.9	0.8	0.7	0.6
Linear	8.5e-3	9.5e-3	8.0e-3	5.0e-3	1.0e-1
Cubic	1.8e-4	1.8e-3	4.0e-2	NaN	Nan
Limited	1.8e-4	1.8e-3	4.3e-2	1.0e-1	1.7e-1
None	1.9e-4	1.9e-3	4.3e-2	1.0e-1	1.7e-1

TABLE 1 1D Advection Reaction Equation L1 Error Norms CFL = 0.0125

Table 1 shows errors for the advection reaction problem using an even mesh of 1601 points. Different values of the constant c are used. When c is some distance from one, the numerical wave solution starts to lag behind the true wave solution. In all cases, the L1 error norm is calculated at time $t = 1.5$.

Four schemes are shown: linear interpolation, cubic interpolation, limited interpolation and none, which is the standard scheme without using any interpolation. For the limited interpolation, the accuracy is close to that of the underlying numerical scheme. For values of $c \leq 0.7$, the overshoots introduced by cubic interpolation cause the solution to become unstable and the calculation to fail. Using linear interpolation surprisingly improves accuracy in some cases, most probably as it adds extra diffusion due to the interpolation error.

6.2 | 3D Test Problems

Experiments to test the interpolation accuracy for 3D problems were conducted on a single node with two Sandy Bridge processors (Intel(R) Xeon(R) CPU E5-2680), each with 8 cores and 90GB of RAM. ULFM version 2.0 was first built using GCC compiler version 6.1.0. Uintah was compiled using MPI wrappers provided by ULFM. Both builds used optimization flag -O3. ULFM was built by disabling the support for MPI_THREAD_MULTIPLE. Uintah was compiled with a flag -fopenmp to provide OpenMP support. OpenMP pragmas are used in both the timestepping code to simulate Burgers' equation and the Advection Reaction equation. They are also used in the interpolation routines to improve performance through data parallelism.

Simulations were run for different combinations of mesh points (ranging from 12^3 to 96^3), different numbers of ranks (4 to 64 by oversubscribing cores), different numbers of sequential failures (from 1 to 5) and different timesteps at which failure is induced. At any given time, half the ranks (odd or even) are killed simultaneously by raising the signal sigkill, and the remaining ranks take over execution. Errors in four cases - timestepping without failure, linear interpolation, ENO, and LENO - were measured and compared against the exact solutions.

Two problems were used to evaluate the accuracies of interpolants. The first is a Burgers' equation problem that illustrates the accuracy differences from the different interpolation approaches. The second problem is a 3D version of the 1D advection-reaction that suffers from catastrophic failures when unconstrained cubic interpolation is used.

6.2.1 | Burgers' Equation:

The viscous Burgers' equation in 1D for $u = \phi(x,t)$ is given by

$$\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = v\frac{\partial^2 \phi}{\partial x^2} \tag{6}$$

where v is the viscosity of the medium. The function $\phi(x,t)$ used for an exact solution for a given timestep at a given location is given by

$$\phi(x,t) = \frac{0.1 + 0.5 * e^{\frac{d-f}{\nu}} + e^{\frac{d-g}{\nu}}}{1 + e^{\frac{d-f}{\nu}} + e^{\frac{d-g}{\nu}}}$$

where d, f and g are calculated as
$a = 0.05*(x - 0.05 + 4.95*t)$, $b = 0.25*(x - 0.5 + 0.750*t)$, $c = 0.5*(x - 0.375)$ and $d = min(a,b,c)$.
if d == a then f = b and g = c
else if d == b then f = a and g = c
else if d == c then f = a and g = b.
The 1D Burgers' equation can be extended to 3D (see Appendix A) for $u = \phi(x,t)\phi(y,t)\phi(z,t)$ as

$$\frac{\partial u}{\partial t} + \phi(x,t)\frac{\partial u}{\partial x} + \phi(y,t)\frac{\partial u}{\partial y} + \phi(z,t)\frac{\partial u}{\partial z} = \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) \quad (7)$$

The initial and boundary conditions are given by

$$u(x,y,z,t) = \phi(x,t)\phi(y,t)\phi(z,t)$$

Figures 2(a) and 2(b) show the results of using the different interpolation methods for Burgers' equation for viscosity

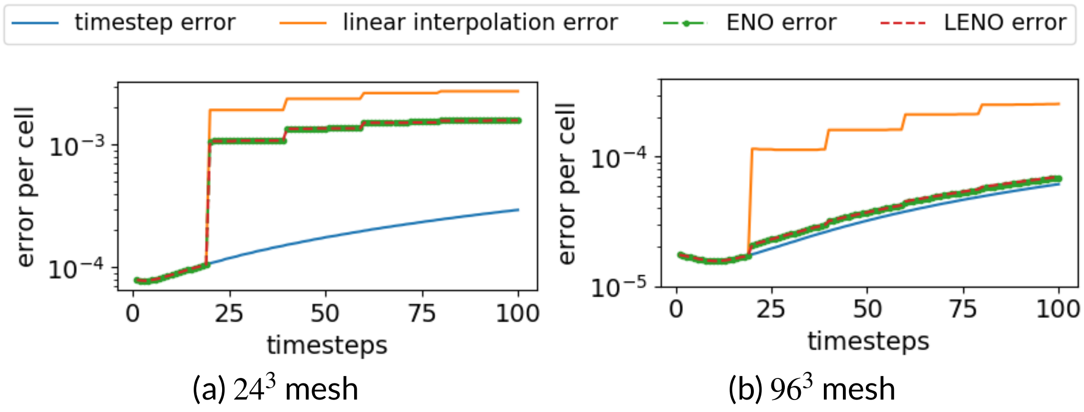


FIGURE 2: Interpolation accuracy for Burgers' equation with viscosity of 0.01

$\nu = 0.01$ and domain sizes of 24^3 and 96^3 meshes, respectively. The experiment spawned 32 MPI ranks and ran for 100 timesteps, with half the ranks failing simultaneously at the 20th, 40th, 60th and 80th timesteps, respectively. Thus, only two ranks remained at the end. As the number of ranks was reduced, CPU cores remained unused. The sudden increase in the error indicates the time at which failure occurred. As the domain resolution increases, the accuracy of the ENO and LENO interpolants improves faster than that of linear interpolation. For the 24^3 domain, the ENO interpolation accuracy was 1.7x times better than that of linear interpolation, but for the 96^3 domain, ENO interpolation had 3.7x times better accuracy than linear interpolation. Although the ENO method performed better than linear interpolation, it caused overshoots in the numerical solution. The numbers of overshoots for cases shown in Figures 2(a) and 2(b) were 17,112 and 25,475, respectively. However, the impact of overshoots is to some extent compensated for by the small value of the timestep and is not significantly reflected in the error.

Using the LENO interpolant eliminates these overshoots by discarding the terms causing overshoots. The LENO approach marginally affects the accuracy. In the case of the 24^3 domain, LENO improved the accuracy over ENO by 2% but degraded the accuracy by 2% for the 96^3 domain. Similar results were obtained for a three-dimensional heat problem, but are omitted for reasons of brevity.

6.2.2 | 3d Advection-Reaction Equation:

The advection-reaction equation used in the one-dimensional test can be extended to three space dimensions as

$$\frac{\partial u}{\partial t} + \frac{\partial u}{\partial x} + \frac{\partial u}{\partial y} + \frac{\partial u}{\partial z} = 40(c-1)u(1-u) \quad (8)$$

The boundary conditions at $x = 0, y = 0$ and $z = 0$ and the initial condition are given by

$$u(x,y,z,t) = \phi((x+y+z)/3,t) \text{ where } \phi(x,t) \text{ is given by } u(x,t) \text{ in equation 5} \quad (9)$$

c	1	0.9	0.8	0.7	0.6	0.5	0.4	0.3	0.2	0.1
None	2.40E-02	1.08E-01	2.13E-01	2.77E-01	2.81E-01	2.31E-01	1.55E-01	8.44E-02	3.61E-02	1.01E-02
Linear	2.44E-02	1.13E-01	2.26E-01	2.94E-01	2.98E-01	2.44E-01	1.62E-01	8.73E-02	3.67E-02	1.01E-02
ENO	2.41E-02	1.08E-01	2.14E-01	2.77E-01	NaN	NaN	NaN	NaN	3.66E-02	1.02E-02
LENO	2.41E-02	1.08E-01	2.15E-01	2.82E-01	2.88E-01	2.41E-01	1.56E-01	8.44E-02	3.65E-02	1.02E-02

TABLE 2 3D Advection reaction equation errors compared to the exact solution

and $(x,y,z,t) = [0,2]x[0,2]x[0,2]x(0,1.5]$. In this case, 32 MPI ranks were used, with half of the ranks crashing at the 100^{th}, 200^{th}, 300^{th}, and 400^{th} timestep. A domain size of 24^3 mesh points is used for $(x,y,z) \in [0,2]x[0,2]x[0,2]$. Table 2 compares errors of the numerical solution using various interpolation schemes, namely linear interpolation, ENO and LENO to the exact solution, whereas "None" represents the error without any failures/interpolation schemes. For this equation, overshoots and undershoots caused by using the ENO method lead to infinite values of the solution for $0.3 \leq c \leq 0.6$. However, LENO methods avoid such errors, which confirms that the accuracy results for the one-dimensional problem translate into three space dimensions. Similar results were obtained for the Leveque-Yee model problem (14) in one and three spatial dimensions.

6.3 | Experiments to measure scalability

Preliminary strong scaling experiments were conducted to compare the performance of the new ABFT approach with checkpointing using 128 nodes of the Quartz cluster at Lawrence Livermore National Laboratory (LLNL) (48). Uintah has its own checkpointing and restart functionality. Users can choose a checkpointing interval and the type of I/O process among (i) I/O per process, (ii) I/O per N processes (i.e., one process collects output from N processes and performs I/O) and (iii) using the PIDX (49, 50, 51) high-performance I/O library. For a small node count, I/O per process performs as well as PIDX on the Lustre file system, which is used by the Quartz cluster (52). For the modest size node counts used in this study, Uintah's built-in I/O per process is used for checkpoint and restore and performs well in comparison to more sophisticated approaches such as PIDX (49, 50, 51).

Each node of Quartz is equipped with 36 cores of the Intel Xeon E5-2695 v4 processor(s) and 128 GB RAM. ULFM and Uintah were compiled in the same way as described in the accuracy experiments using GCC 6.1.0. Experiments were conducted using the three-dimensional Burgers' equation with a fine mesh of size 256^3 points and a coarse mesh of 128^3 points (i.e., the refinement ratio of two). Both grids were divided into 4096 patches (total 8192 patches). Thus, the fine patch size was 16^3 and the coarse patch size was 8^3.

Failure Model: Strong scaling starting from two nodes up to 128 nodes was carried out with one rank per node. During every run, half of the ranks (odd or even) raise "sigkill" and simultaneously crash. Soon after the crash, MPI detects an error and the custom error handler kicks in. Unlike the model used to verify the accuracy, failures occur at only one instance, i.e. at the 20^{th} timestep. Thus, for every crash, half of the data (i.e., 2048 patches) is interpolated by the surviving ranks. Although this is not a realistic failure model with half of the nodes failing at the same time, the point of considering this the worst-case scenario was to demonstrate the effectiveness of this approach with extreme failure rates. Different studies have been conducted in the past to categorize failure rates and types (53, 54, 55). For example, Meneses (54) shows that only 29% of the software failures during 2014 affected more than four nodes, and only 2% of the hardware failures affected four or more nodes on the Titan supercomputer. However, "mean time between failures" (MTBF) of Titan is around 27 hours (54). On the other hand, the DARPA ExaScale Computing Study (56) suggests that the MTBF at exascale could be as frequent as 35-39 minutes. This 45-fold increase in the failure rate indicates the possibility of multiple instances of failures within the lifetime of a job. Some of these failure instances might have concurrent node failures as well. Thus, our failure model assumes that half of the nodes crash simultaneously to prepare Uintah for the worst case scenario. Figure 3 shows the overhead incurred by the resiliency tasks during normal timesteps. The "not resilient" plot

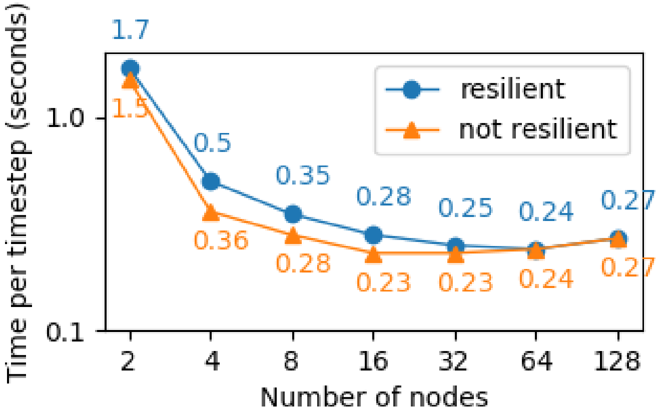
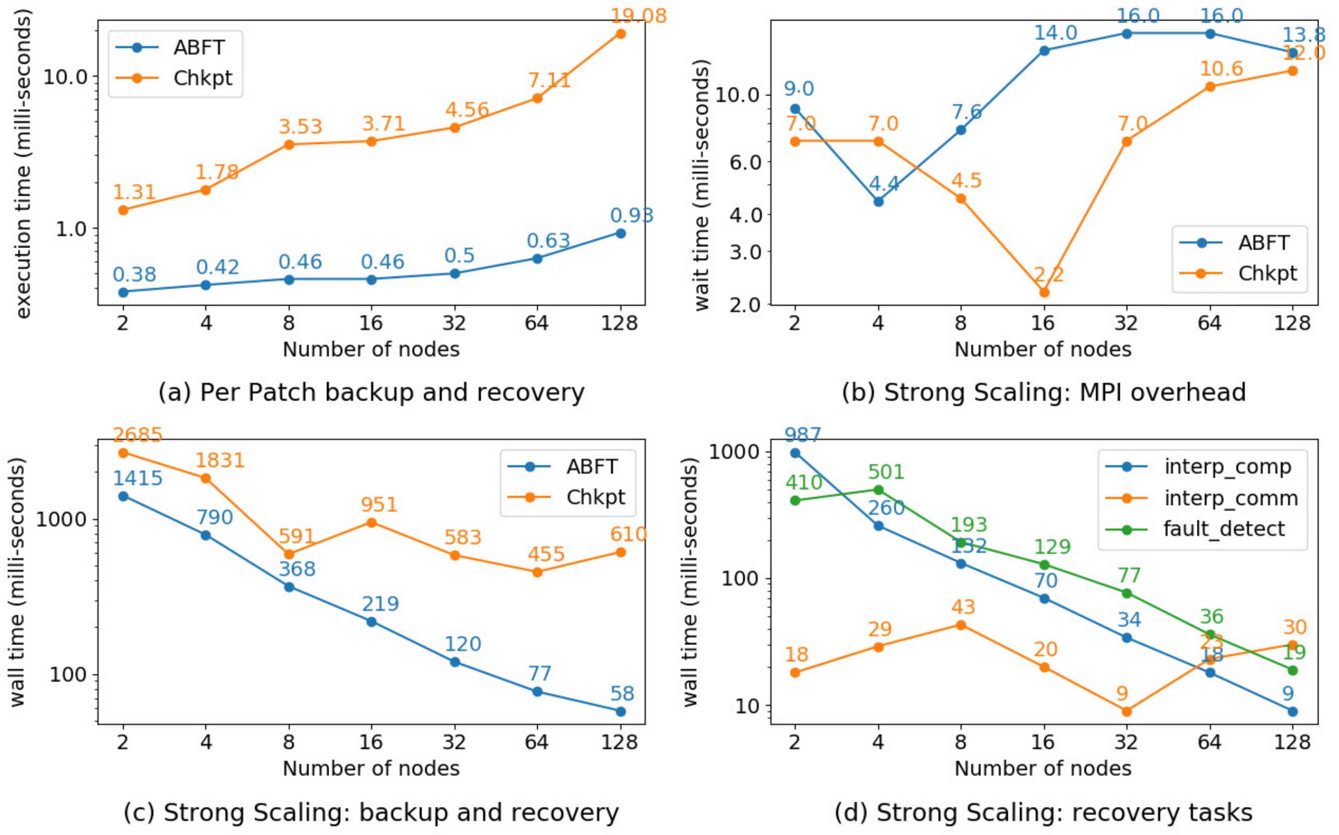


FIGURE 3: Overhead of resiliency on LLNL Quartz cluster

shows time per timestep when resiliency is turned off. The “resilient” plot shows wall time per timestep for the resilient approach and includes tasks to generate coarse patches and to perform the coarse patch exchanges between neighbors. The plot does not include the recovery phase, which is analyzed later. The difference between the two plots shows that the overhead of resilience is around 38% for four nodes when each node processes 1024 patches. As the number of nodes increases, the number of patches per node decreases, resulting in less overhead. Increasing the node count to 64 and 128, the number of patches per node decreased to 64 and 32, respectively, with the resultant overhead decreasing to less than 10 milliseconds. Typically, Uintah’s domain decomposition strategy usually assigns 1-2 patches per core. For Quartz’s 36 core nodes, using this workload of 32 or 64 patches per node matches Uintah’s domain decomposition scheme for realistic simulations.

To analyze the performance of resilience, measurements are made of (i) the time to coarsen patches, (ii) MPI communications during regular timesteps and (iii) the time to detect and recover from an error and interpolate lost data during the recovery timestep. Similar experiments were repeated using checkpoint/restarts instead of using ABFT resilience. The time spent in IO tasks during checkpointing and in recovery was also measured. Again, the MPI communication time was also measured for the timesteps in which checkpointing does not take place. These timing values were also used to deduce the per-patch overhead in both methods and to measure the communication overheads due to coarsening, patch exchange and the per-patch recovery time. Three metrics were used to compare the performance



(a) Per Patch backup and recovery

(b) Strong Scaling: MPI overhead

(c) Strong Scaling: backup and recovery

(d) Strong Scaling: recovery tasks

FIGURE 4: Performance comparison on LLNL Quartz cluster

of ABFT-based resiliency with checkpointing within Uintah.

1. Per patch per timestep time for backup and recovery: For ABFT-based resiliency, this metric is a sum of the computation and communication wait times needed to execute coarsening tasks per timestep, the time ULFM takes to detect a set of failed ranks (in the case of simultaneous failures), and the subsequent recovery and interpolation operations. For the checkpointing approach, this metric is the sum of the time taken by one checkpoint operation (equivalent to one coarsening and exchange task of ABFT during one timestep) and the time

taken for one recovery from the checkpoint. Coarse patches are exchanged at every timestep, whereas checkpointing takes place after every four timesteps. Both these frequencies can be adjusted depending upon the problem to be solved. For a fair comparison, only one checkpoint-restart was timed and compared against one coarsening + exchange and ABFT recovery. This approach allows for a comparison of the raw execution times. Figure 4(a) shows that ABFT-based solution performed 3.4x faster than checkpointing on two nodes, and the gap increased with the number of nodes. ABFT performance was almost linear up to 64 nodes, after which it starts to degrade. However, checkpointing performance degrades as the number of nodes increases. At 128 nodes, the performance boost from ABFT becomes 20X.

2. MPI overhead exchange of coarse patches: In the ABFT approach, the coarsening and exchange tasks execute more frequently than recovery. These tasks should have minimal overhead. The coarse task did not need any communication and was observed to be scalable from 125 ms for two nodes to 1.8 ms for 128 nodes (not shown in the chart for simplicity). However, exchanging coarse patches adds a small amount of overhead during MPI communication, as shown in Figure 4(b). The difference between the MPI wait time of Uintah when coarse patches are exchanged, and the MPI wait time when coarse patches are not calculated and not exchanged, shows the MPI overhead due to the coarsening tasks.

   Uintah is designed to overlap computation and communication. However, if coarse patches are exchanged in an overlapped fashion, there is a risk of failure before the patch exchange is completed, and the recovery will become impossible. To avoid this situation, no other tasks are allowed to execute before coarse patch exchange is completed during every timestep. A communication overhead is visible in Figure 4(b), even for eight nodes where each rank has enough computation to effectively hide this exchange. As the number of nodes increases, the number of coarse patches exchanged per node decreases, and the overhead decreases from 700% for 16 nodes to 10% for 128 nodes. This overhead can be further reduced by introducing a timestep lag during the exchange, and thus the recovery routine will have to execute one extra timestep. At 128 nodes, communication starts dominating computation (for the entire calculation and not just coarsening tasks). The effect can be seen in Figures 4(a) and (c) as the number of nodes increases to 128. This strong scaling pattern of coarsening computations and MPI overheads concurs with Figure 3, where the resiliency overhead decreases as the number of nodes increases.

3. Strong scaling of backup and recovery: This metric compares the wall clock time of similar operations involved in the per patch metric, but now the wall clock time to coarsen/recover or checkpoint/recover all patches is measured. This metric demonstrates strong scaling of both approaches. Figure 4(c) shows that ABFT scales better than checkpointing. There is one seeming contradiction between Figures 4(a) and (c). The speed-up in Figure 4(a) is twice that of the speed-up shown in Figure 4(c). The reason for the apparent paradox is the number of ranks is halved after the recovery from a failure. In other words, for 128 nodes, one cannot expect the wall clock time to be 0.93 (per patch time) * 4096 (number of patches) / 128 (number of nodes) = 29.76 milliseconds. After the failure, only 64 nodes execute the program and, hence, the expected wall time will be 0.93 (per patch time) * 4096 (number of patches) / 64 (number of nodes) = 59 milliseconds, which is close to the observed value of 58 milliseconds. Experiments also demonstrated that, in the case of multiple simultaneous failures, the wall clock time to recover from one failure remains the same as the wall clock time to recover from 64 failures, as recovery takes place in parallel. Figure 4(d) gives a breakdown of the wall clock time in three categories - error detection + state recovery, interpolation and the MPI wait time for halo exchange required by interpolation tasks. Due to the use of OpenMP enabled loops, interpolation tasks execute faster than error detection + state recovery, which is a single-threaded task.

Comparison to high-performance checkpointing: PIDX is a high-performance I/O library built for high-performance computing, and performs better than the state-of-the-art I/O libraries (49, 50, 51). Uintah with PIDX was able to utilize 80% of the theoretical disk bandwidth on MIRA (52). Compared to naive per process I/O, PIDX achieved a speed-up of 2x to 10x as the number of processes increased from 8192 to 262,890 (52). However, the speed-up on Edison was limited to 0.7x to 1.3x as the number of cores increased from 1024 to 8192. This difference in performance occurs because Edison's Lustre filesystem performs better while handling large numbers of files compared to MIRA's GPFS file system (52). Charm++ has a resiliency capability using in-memory checkpointing on the "buddy" process, which provided about 100x speed-up over checkpointing (57). Zheng (58) further optimized the performance that resulted in checkpointing and restart finishing in a few milliseconds. This performance is as good as the ABFT performance achieved by resilient Uintah. Dong (59) was able to reduce the checkpointing overhead to $2-3\%$ using non-volatile RAM (NVRAM) for checkpointing. In a similar work, Kannan (60) reported checkpointing overhead to 6% of the total run time.

In spite of these advances, checkpointing has it own challenges. Its performance also depends on environmental factors using shared resources. Moody (61) experienced an increase from 3.5 minutes (for 4096 processes on 512 nodes) to 1.5 hours (for 8192 processes on 1024 nodes) for checkpointing. The drastic slowdown was due to the load from other jobs and not the increased number of ranks (61). The MTBF increased from 1.5 hours to 1.5 days when the Scalable Checkpoint/Restart (SCR) library was used instead of traditional checkpointing on the Atlas cluster at Lawrence Livermore National Laboratory (61). Nevertheless, it will be interesting to compare the performance of Resilient Uintah with these techniques at scale.

7 | CONCLUSION

The combination of ULFM, AMR, and interpolation used here has been shown to be a faster recovery method than using the standard approach of checkpointing. The recovery routines were able to recover from repeated simultaneous failures until only one rank remained. Furthermore, recovery is treated as just another task that requires only a load-balancing step, which makes this approach more flexible without placing an extra burden on the runtime system. This approach has the potential to completely avoid delays and interruptions, because survivor ranks can always continue the execution of other tasks while recovery is in progress.

Using ULFM to detect failed ranks and using interpolation for patch recovery provides a lightweight approach to ensure that Uintah is more resilient in the face of future generations of architectures with significantly higher MTBF.

The results presented here demonstrate the effectiveness of using limited ENO over linear interpolation and ENO methods to recover data lost by node failures. The Limited ENO method provides physically meaningful solution values that do not cause difficulties with associated physics routines. The combination of AMR with physically appropriate interpolation methods along with fault-tolerant ULFM and simple recovery tasks ensures that the ABFT approach for Uintah is fault-tolerant for future architectures.

8 | FUTURE WORK

Many opportunities exist to optimize the current implementation and compare it against other resiliency approaches.

The dynamic load-balancing capability of Uintah can minimize any load imbalance created by the recovery process. Overlapping the communication and computation while exchanging coarse patches will significantly reduce the resiliency overhead - especially at a smaller node count. An alternative to AMR and interpolation is the compression and duplication of data to neighboring nodes, which can be transferred and uncompressed to recover from failures. The ABFT approach has the advantage of strong scaling and communication only to neighboring nodes. Hence, it will be interesting to compare the performance of ABFT at large scale with the latest checkpointing methods that use non-volatile RAM and in-RAM.

Future work may explore an option to allocate extra “reserve” nodes and ranks at the beginning, and to manipulate the MPI communicator to maintain a fixed number of “active” ranks. On failure, reserve ranks can replace failed ranks. The challenge for this approach will be passing the active state of the simulation to the reserve ranks. Finally, increasing the number of nodes for the exchange of coarse patches from the current design of a single node to two or more nodes will allow a greater level of fault tolerance and flexibility.

9 | ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. 1337145. The authors are grateful to Alan Humphrey for his help with Uintah.

References

[1] Exascale Computing The ASCAC Subcommittee. The Opportunities and Challenges of Exascale Computing. *Summary Report of the Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee.* 2010;.

[2] Dauwe D., Pasricha S., Maciejewski A. A., Siegel H. J.. An Analysis of Resilience Techniques for Exascale Computing Platforms. In: *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* , :914-923; May 2017.

[3] Dauwe D., Pasricha S., Maciejewski A. A., Siegel H. J.. A Performance and Energy Comparison of Fault Tolerance Techniques for Exascale Computing Systems. In: *2016 IEEE International Conference on Computer and Information Technology (CIT)* , :436-443; Dec 2016.

[4] Peterson B., Humphrey A., Schmidt J., Berzins M.. Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs. Awarded Best Paper. In: *Proceedings of the Third International Workshop on Extreme Scale Programming Models and Middleware - ESPM2'17* , ACM; 2017.

[5] Peterson B., Xiao N., Holmen J., et al. *Developing Uintah's Runtime System For Forthcoming Architectures. Refereed paper presented at the RESPA 15 Workshop at SuperComputing 2015 Austin Texas..* : SCI Institute; 2015.

[6] Berzins Martin, Beckvermit Jacqueline, Harman Todd, et al. Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices. *SIAM Journal on Scientific Computing.* 2016;38(5):S101–S122.

[7] Humphrey A., Sunderland D., Harman T., Berzins M.. Radiative Heat Transfer Calculation on 16384 GPUs Using a Reverse Monte Carlo Ray Tracing Approach with Adaptive Mesh Refinement. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* , :1222-1231; May 2016.

[8] Meng Qingyu, Humphrey Alan, Schmidt John, Berzins Martin. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* , :96:1–96:12ACM; 2013.

[9] Holmen J. K., Humphrey A., Sutherland D., Berzins M.. Improving Uintah's Scalability Through the Use of Portable Kokkos-Based Data Parallel Tasks. In: *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact* , no. 27 in PEARC17:27:1–27:8; 2017.

[10] Dubey Anshu, Mohapatra Prateeti, Weide Klaus. Fault Tolerance Using Lower Fidelity Data in Adaptive Mesh Applications. 2013;:3–10.

[11] Bland W. User Level Failure Mitigation in MPI. *Euro-Par 2012: Parallel Processing Workshops.* 2013;:499–504.

[12] Berzins M., Capon P.J., Jimack P.K.. On Spatial Adaptivity and Interpolation When Using the Method of Lines. *Applied Numerical Mathematics.* 1998;26:117–134.

[13] Berzins M.. Adaptive Polynomial Interpolation on Evenly Spaced Meshes. *SIAM Review.* 2007;49(4):604–627.

[14] LeVeque Randall J, Yee Helen C. A study of numerical methods for hyperbolic conservation laws with stiff source terms. *Journal of computational physics.* 1990;86(1):187–210.

[15] LIGHT D., Durran D.. Preserving Nonnegativity in Discontinuous Galerkin Approximations to Scalar Transport via Truncation and Mass Aware Rescaling (TMAR). *Monthly Weather Review.* 2016;144:4771–4786.

[16] Huber M, Gmeiner B, Rüde U, Wohlmuth B. Resilience for Massively Parallel Multigrid Solvers. *SIAM Journal on Scientific Computing.* 2016;38(5):S217-S239.

[17] Agbaria Adnan, Friedman Roy. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. In: *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing* , HPDC '99:31–IEEE Computer Society; 1999; Washington, DC, USA.

[18] Bosilca G., Bouteiller A., Cappello F., et al. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In: *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing* , :29-29; 2002.

[19] J. Dongarra, T. Herault, Y. Robert. Fault Tolerance Techniques for High-Performance Computing. In: J. Dongarra, T. Herault, Y. Robert, eds.*Fault-Tolerance Techniques for High-Performance Computing* , Springer; 2015.

[20] Boutellier A.. Fault Tolerant MPI. In: *Fault-Tolerance Techniques for High-Performance Computing* , Springer; 2015.

[21] Cappello F, Geist A, Gropp W, Kale S, Kramer B, Snir M. Toward Exascale Resilience: 2014 Update. *Supercomputing Frontiers and Innovations: an International Journal.* 2014;1(1):5–28.

[22] Hargrove P, Duell J. Berkeley lab checkpoint/restart (blcr) for Linux clusters. *Journal of Physics: Conference Series.* 2006;46:494.

[23] Shi Xuanhua, Pazat Jean-Louis, Rodriguez Eric, Jin Hai, Jiang Hongbo. Adapting grid applications to safety using fault-tolerant methods: Design, implementation and evaluations. *Future Generation Computer Systems.* 2010;26(2):236–244.

[24] Wang Chao, Mueller Frank, Engelmann Christian, Scott Stephen L. Hybrid checkpointing for MPI jobs in HPC environments. In: *Parallel and Distributed Systems (ICPADS), 2010 IEEE 16th International Conference on* , :524–533IEEE; 2010.

[25] Islam Tanzima Zerin, Mohror Kathryn, Bagchi Saurabh, Moody Adam, De Supinski Bronis R, Eigenmann Rudolf. McrEngine: a scalable checkpointing system using data-aware aggregation and compression. *Scientific Programming.* 2013;21(3-4):149–163.

[26] Bronevetsky Greg, Marques Daniel, Pingali Keshav, McKee Sally, Rugina Radu. Compiler-enhanced incremental checkpointing for openmp applications. In: *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on* , :1–12IEEE; 2009.

[27] Hussain Zaeem, Znati Taieb, Melhem Rami. Partial Redundancy in HPC Systems with Non-uniform Node Reliabilities. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* , SC '18:44:1–44:11IEEE Press; 2018; Piscataway, NJ, USA.

[28] Sato Kento, Maruyama Naoya, Mohror Kathryn, et al. Design and modeling of a non-blocking checkpointing system. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* , :19:1–19:10IEEE Computer Society Press; 2012.

[29] Huang Kuang-Hua, Abraham Jacob. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers.* 1984;100(6):518–528.

[30] Chen Zizhong, Dongarra Jack. Algorithm-based checkpoint-free fault tolerance for parallel matrix computations on volatile resources. In: *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International* , :10–ppIEEE; 2006.

[31] Chen Zizhong, Dongarra Jack. Algorithm-based fault tolerance for fail-stop failures. *IEEE Transactions on Parallel and Distributed Systems.* 2008;19(12):1628–1641.

[32] Davies Teresa, Chen Zizhong. Correcting soft errors online in LU factorization. In: *Proceedings of the 22nd international symposium on High-performance parallel and distributed computing* , :167–178ACM; 2013.

[33] Davies Teresa, Karlsson Christer, Liu Hui, Ding Chong, Chen Zizhong. High performance linpack benchmark: a fault tolerant implementation without checkpointing. In: *Proceedings of the international conference on Supercomputing* , :162–171ACM; 2011.

[34] Chien A, Balaji P, Dun N, et al. Exploring versioned distributed arrays for resilience in scientific applications: global view resilience. *The International Journal of High Performance Computing Applications.* 2016;.

[35] Fang A, Cavelan A, Robert Y, Chien A A.. Resilience for Stencil Computations with Latent Errors. In: *International Conference on Parallel Processing (ICPP)* , ; August 2017.

[36] Dubey A., Fujita H., Graves D.T., Chien A., Tiwari D.. Granularity and the Cost of Error Recovery in Resilient AMR Scientific Applications. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* , SC '16:42:1–42:10IEEE Press; 2016; Piscataway, NJ, USA.

[37] Fagg G, Dongarra J. FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world.. *7th European PVM/MPI Users Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface.* 2000;:346–353.

[38] Varma Jyothish, Wang Chao, Mueller Frank, Engelmann Christian, Scott Stephen L.. Scalable, Fault Tolerant Membership for MPI Tasks on HPC Systems. In: *Proceedings of the 20th Annual International Conference on Supercomputing* , ICS '06:219–228ACM; 2006; New York, NY, USA.

[39] Rizzi Francesco, Morris Karla Vanessa, Cook B, et al. *Performance Scaling Variability and Energy Analysis for a Resilient ULFM-based PDE Solver..* : Sandia National Lab.(SNL-CA), Livermore, CA (United States); 2016.

[40] Luitjens J., Berzins M.. Improving the Performance of Uintah: A Large-Scale Adaptive Meshing Computational Framework. In: *Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS10)* , :1–10; 2010.

[41] Gasca Mariano, ThomasSauer . On the history of multivariate polynomial interpolation. *Journal of Computational and Applied Mathematics.* 2000;122:23–35.

[42] Narumi S.. Some formulas in the theory of interpolation of many independent variables. *Tohoku Math. J..* 1920;18:309–321.

[43] McCorquodale Peter, Colella Phillip, Grote David P, Vay Jean-Luc. A node-centered local refinement algorithm for Poisson's equation in complex geometries. *Journal of Computational Physics.* 2004;201(1):34–60.

[44] Martin Daniel F, Colella Phillip. A cell-centered adaptive projection method for the incompressible Euler equations. *Journal of computational Physics.* 2000;163(2):271–312.

[45] Harten A., Engquist B., Osher S., Chakravarthy S. R.. Uniformly high-order accurate essentially nonoscillatory schemes. *III, J. Comput. Phys.* 1987;49(4):231–303.

[46] Shu Chi-Wang. High order ENO and WENO schemes for computational fluid dynamics. In: *High-order methods for computational physics* , Springer 1999 (pp. 439–582).

[47] Ahmad I., Berzins M.. MOL Solvers for Hyperbolic PDEs with Source Terms. *Mathematics and Computers in Simulation.* 2001;56:1115–1125.

[48] https://hpc.llnl.gov/hardware/platforms/Quartz. ;.

[49] Kumar S., Vishwanath V., Carns P., et al. PIDX: Efficient Parallel I/O for Multi-resolution Multi-dimensional Scientific Datasets. In: *Proceedings of The IEEE International Conference on Cluster Computing* , :103–111; 2011.

[50] Kumar S., Vishwanath V., Carns P., et al. Efficient data restructuring and aggregation for I/O acceleration in PIDX. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* , :50:1–50:11IEEE Computer Society Press; 2012.

[51] Kumar S., Hoang D., Petruzza S., Edwards J., Pascucci V.. Reducing Network Congestion and Synchronization Overhead During Aggregation of Hierarchical Data. In: *2017 IEEE 24th International Conference on High Performance Computing (HiPC)* , :223-232; 2017.

[52] Kumar S., Humphrey A., Usher W., et al. Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation. In: *Supercomputing Frontiers* , :219–240Springer International Publishing; 2018.

[53] Di Martino Catello, Kalbarczyk Zbigniew, Iyer Ravishankar K, Baccanico Fabio, Fullop Joseph, Kramer William. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* , :610–621IEEE; 2014.

[54] Meneses Esteban, Ni Xiang, Jones Terry, Maxwell Don. Analyzing the interplay of failures and workload on a leadership-class supercomputer. *computing.* 2015;2(3):4.

[55] Gupta Saurabh, Patel Tirthak, Engelmann Christian, Tiwari Devesh. Failures in large scale systems: long-term measurement, analysis, and implications. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* , :44ACM; 2017.

[56] Bergman Keren, Borkar Shekhar, Campbell Dan, et al. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep.* 2008;15.

[57] Zheng Gengbin, Shi Lixia, Kalé Laxmikant V. FTC-Charm++: an in-memory checkpoint-based fault tolerant runtime for Charm++ and MPI. In: *2004 ieee international conference on cluster computing (ieee cat. no. 04EX935)* , :93–103IEEE; 2004.

[58] Zheng Gengbin, Ni Xiang, Kalé Laxmikant V. A scalable double in-memory checkpoint and restart scheme towards exascale. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)* , :1–6IEEE; 2012.

[59] Dong Xiangyu, Muralimanohar Naveen, Jouppi Norm, Kaufmann Richard, Xie Yuan. Leveraging 3D PCRAM technologies to reduce checkpoint overhead for future exascale systems. In: *Proceedings of the conference on high performance computing networking, storage and analysis* , :57ACM; 2009.

[60] Kannan Sudarsun, Gavrilovska Ada, Schwan Karsten, Milojicic Dejan. Optimizing checkpoints using nvm as virtual memory. In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing* , :29–40IEEE; 2013.

[61] Moody Adam, Bronevetsky Greg. *Scalable I/O systems via node-local storage: Approaching 1 TB/sec file I/O.* : Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States); 2008.

☐

APPENDIX A: DERIVATION OF 3D BURGERS' EQUATION

$$\begin{aligned}\frac{\partial u}{\partial t} &= \frac{\partial \phi(x,t)}{\partial t}\phi(y,t)\phi(z,t)\\ &+\phi(x,t)\frac{\partial \phi(y,t)}{\partial t}\phi(z,t)\\ &+\phi(x,t)\phi(y,t)\frac{\partial \phi(z,t)}{\partial t}\end{aligned} \tag{A1}$$

Now using equation 6, burger's equations for independent individual dimensions as can be specified as :

$$\frac{\partial \phi(x,t)}{\partial t} = -\phi(x,t)\frac{\partial \phi(x,t)}{\partial x} + \nu\frac{\partial^2 \phi(x,t)}{\partial x^2} \tag{A2}$$

$$\frac{\partial \phi(y,t)}{\partial t} = -\phi(y,t)\frac{\partial \phi(y,t)}{\partial y} + \nu\frac{\partial^2 \phi(y,t)}{\partial y^2} \tag{A3}$$

$$\frac{\partial \phi(z,t)}{\partial t} = -\phi(z,t)\frac{\partial \phi(z,t)}{\partial z} + \nu\frac{\partial^2 \phi(z,t)}{\partial z^2} \tag{A4}$$

Substituting A2, A3, A4 and in 6:

$$\begin{aligned}\frac{\partial u}{\partial t} &= -\frac{1}{\phi(y,t)\phi(z,t)}u\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}\\ &-\frac{1}{\phi(x,t)\phi(z,t)}u\frac{\partial u}{\partial y} + \nu\frac{\partial^2 u}{\partial y^2}\\ &-\frac{1}{\phi(x,t)\phi(y,t)}u\frac{\partial u}{\partial z} + \nu\frac{\partial^2 u}{\partial z^2}\end{aligned}$$

Or alternative form is:

$$\begin{aligned}\frac{\partial u}{\partial t} = &-\phi(x,t)\frac{\partial u}{\partial x} + \nu\frac{\partial^2 u}{\partial x^2}\\ &-\phi(y,t)\frac{\partial u}{\partial y} + \nu\frac{\partial^2 u}{\partial y^2}\\ &-\phi(z,t)\frac{\partial u}{\partial z} + \nu\frac{\partial^2 u}{\partial z^2}\end{aligned}$$

This can be further simplified into:

$$\frac{\partial u}{\partial t} + \phi(x,t)\frac{\partial u}{\partial x} + \phi(y,t)\frac{\partial u}{\partial y} + \phi(z,t)\frac{\partial u}{\partial z} = \nu(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}) \tag{A5}$$