An Efficient Method for Component Failure Resiliency in the Uintah Framework

Aditya Pakki
School of Computing, University of Utah
Salt Lake City, UT, 84112, USA
pakkiadi@cs.utah.edu

ABSTRACT

The problem of dealing efficiently with failures at node and core level during simulations in the Uintah Computational Framework (Uintah) for present and future architectures is addressed through minimal (coarse) patch replication and (re)scheduling of tasks. In this report, we describe the results and observations of adopting these approaches as Uintah's alternate to checkpoint and restart strategy. We characterize the maximum number of cores that can fail in a node without necessitating load balance and determine the ideal time to declare node failure in case of multiple core failures. A novel implementation of bounded cubic interpolation technique triggered during node failures is described. This recovery technique, comparable to linear interpolation in terms of compute time, uses duplicate data patches stored across nodes, requiring an additional space of 12.5%. The accuracy of recovered patches shows that interpolating from coarse data is advantageous, in terms of reduced storage and faster recovery time, than naïve checkpointing.

Keywords

Interpolation, Resiliency, Uintah, Task migration, Parallel

1. INTRODUCTION

The biannual lists of the fastest machines released by top500.org, convey a pattern that clusters of commodity components and Massively Parallel Programming (MPP) systems would be the future of supercomputer architecture. The proposed architecture of next generation pre-Exascale machines, expected to be available earliest by 2018, confirm to this trend of increased number of components (more cores per node, more nodes per rack). Unfortunately, massive increase in component count with a relatively slower increase in reliability characteristics of hardware, decreases the Mean Time Between Failure (MTBF) of the system. Robust components (network interconnect, memory, cores, nodes, disks, etc.) are a necessity for application developers of high performance computing (HPC) community, to take advantage of the increasing compute power and to minimize computation loss due to system failures. It is often remarked that, at Exascale, frequent failure of components and larger volume of checkpoint data would be prohibitive for the computation. Achieving fault tolerance, by techniques other than simple checkpoint and restart is expected to be the way forward for Exascale runtime systems. In this report, we consider an alternate approach to checkpointing, focusing on re-execution of tasks during core failure and replication of coarser patches followed by interpolation to regenerate missing data during node failure.

The cost of solving a numerical problem on a computational mesh depends on the number of points the problem is discretized as well as the number of steps needed to reach the accuracy. Coarse meshing increases the steps to reach the accuracy but uses fewer points and fine meshing increases the computational and communication costs but provides greater accuracy. Adaptive Mesh Refinement (AMR) provides a neat trade-off between the two factors by discretizing the mesh finely at regions with greater detail and coarsely at regions with lesser detail. Uintah, uses Block Structured AMR (BSAMR) to identify regions needing refinement dynamically by setting flags and uses different spatial levels to compute the same problem. Uintah's AMR framework stores the coarse level data even for a fine level mesh in separate hash-based data structures called DataWarehouse. This additional storage of coarse data patches is used to recreate fine patches lost during a node failure using interpolation. Recreating the fine to coarse patches is a straight forward approach of skipping every alternate data point in the fine mesh.

The main contribution of this report is, showing the viability of an alternate technique to checkpoint and recovery within Uintah for fail stop errors, making large scale simulations resilient. The effectiveness of cubic bounded interpolation technique instead of using default data values or linear interpolation is highlighted.

The remainder of this report is organized as follows - Section 2 provides the background for the problem and the corresponding state of the art research, followed by an overview of Uintah with some important terminologies. Core failure recovery in a single node is described in Section 3 and node level failure recovery technique, using interpolation and its mathematical treatment is described in Section 4. The results of the experiments from Sections 3 and 4 are described in Section 5. In section 6, we present our conclusions from these implementations and give our research direction and ideas.

2. BACKGROUND

2.1 Related Work

In a traditional distributed system to deal with failures, there are many components that should be available, which can be broadly divided as failure monitoring, fault detection, failure identification, and fault recovery. While implementing these techniques require constant communication and overlap as in Figure 1, the effectiveness of these techniques in a parallel system requires a paradigm thought shift of tightly coupled system. For example, the idea of monitoring the nodes for failures by performing a ping after a short interval could swamp the network and hinder MPI communication. A missed error detection due to a larger delay in the ping interval, can be disastrous and fails the computation. These trade-off decisions provide some interesting cases to the HPC research community. An alternative strategy is to envision a systems level design

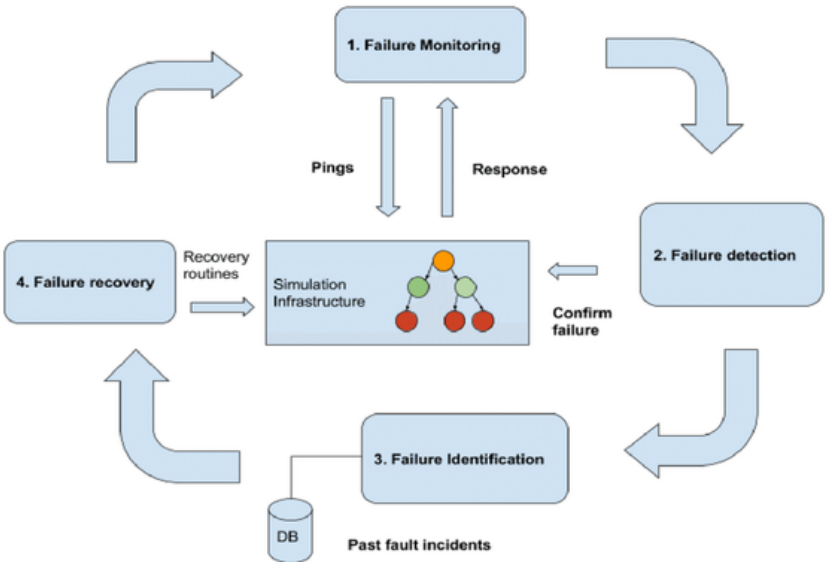

Figure 1: Failure check in a traditional system

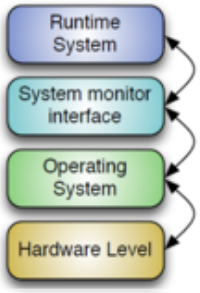

Figure 2: Alternate resilience layers

encompassing resiliency interface between the hardware and software as seen in Figure 2

Few other techniques that attempt to solve the problem of resiliency include message logging, replication of process execution, failure prediction, failure avoidance, hybrid approaches involving one of the above approaches with checkpointing. An alternate technique that minimizes periodic checkpointing is the use of algorithm based fault tolerance techniques (ABFT) [10]. These methods use the properties of the computational algorithm in linear algebra to regenerate the lost data in case of failures. Though ABFT is an interesting approach, the cost of designing resilient algorithms is very expensive [9]. Building a resilient infrastructure can be a cheaper alternative to modifying algorithms.

There are many interfaces and libraries written by research groups attempting to solve the problem of frequent failures. Many variations of checkpointing [7, 8, 11, 14, 21, 23] have been published that include asynchronous checkpointing, scalable checkpointing, hybrid checkpointing or checkpointing only critical data. All these measures, try to reduce the volume of data to be check-pointed. However, most of these projects checkpoint to the disk and the I/O bounds of disk writes have not been reduced significantly. Checkpointing at asynchronously timed intervals, forces the processors to communicate and agree on the common time of checkpoint.

We believe, our paper is next to the paper by Dubey et al [13] in using the coarser mesh values stored by AMR to regenerate the lost data during failures. While the authors propose using interpolation of previous steps data, we have used interpolation to regenerate the data, when the current timesteps have failures. This obviates the need for explicit failure detection and identification stages. The other advantage we foresee, is our technique avoids processors staying idle during recovery. The study by Moody et al [19] claim that over 85% of the node failures impacted a single node, implying that the time spent on re-execution of tasks on healthy nodes is wasted. Currently, there are no studies to our knowledge that have identified cases of single or multi core failure within a single node. We believe that advancement in the node and on-node accelerator architectures, would better classify failures to a particular core instead of the entire node.

2.2 Uintah Framework

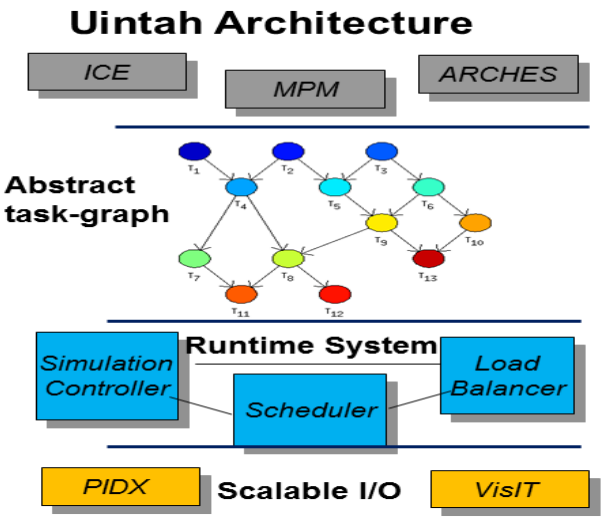

Figure 3: Uintah Framework in a nutshell

[1] The Uintah open-source software has been used for many different types of problems involving fluids, solids and fluid-structure interaction problems. Uintah framework (Figure 3) consists of a set of parallel software components and libraries that facilitate solution of partial differential equations on structured adaptive mesh refinement grids, using hundreds to thousands of processors. Uintah's strength is its use of task graph representation to describe computation and communication and a component based approach that enables independent development of infrastructure and algorithms. To be scalable on some of the fastest parallel computers such as (Stampede, Mira, Titan, Vulcan, etc), few of the advances in Uin-

[1] *Most of the material in subsection 2.2 and section 3 has been borrowed from a peer reviewed accepted workshop submission [20], where the report author was a co-author.*

tah framework include: scalable adaptive mesh refinement [16], novel load balancing [15] implementing out-of-order execution of task graphs [18], using Pthreads with MPI as Uintah MPI+X model, lock-free shared memory approach for efficient data access, using on-node accelerators, using multiple task queues, building a unified scheduler [17].

Below are few of the terms used frequently in this report:

Grids, Levels, Patches, Cells: Solving a computation problem in Uintah (see Figure 4) is possible by declaring various conditions in an input .ups file. A Grid can be considered as the overview of the problem. Each grid can contain multiple or a single level, if AMR is set or not. Each level may contain multiple patches, and patches are then assigned to unique processors. Each patch is further divided into hexhedral structures called cells. Patches are zero indexed as three integer array in each direction, as well as relative indexed per level starting at x, followed by y and z directions. These cells store the physical properties of the grid such as temperature, gravity, pressure or residual etc. Cells can be cell centered or face centered or node centered, containing one, six or eight variables respectively. These variables are indexed by name, type and patch id of the patch to which it belongs.

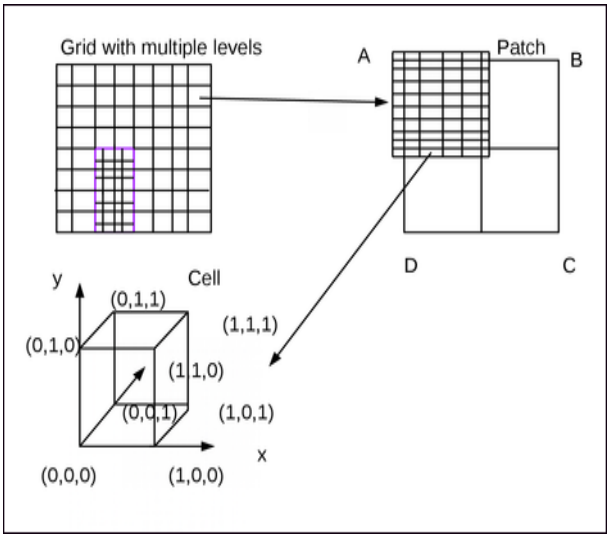

Figure 4: Computational grid, levels, patches and cells

SimulationController: Uintah framework has many simulation algorithms, with ICE being the compressible multi material CFD formulation, particle based Material Point Method(MPM) and combined fluid structure interaction algorithms (MPMICE) and Arches, used to model heat flux and radiation from coal combustion . Apart from these algorithms, there are many components such as Regridders, Schedulers, Load Balancers. Each of these components, have their specific properties which form the basis of any simulation. Simulation Controller manages the interaction between all these various components, keeping an eye on load balance, to attain ideal weak and strong scaling on larger machines.

DataWarehouse: A hash based structure that stores various values used during a simulation step. Read and write access to DataWarehouse is by get() and put() functions. Data from previous step is typically used to compute the newer value which is then stored in the new DataWarehouse. This structure is automatically stored in an output .uda file by the DataArchiver at a frequency specified in the input .ups file. In case of a failure, the simulation will be restarted from the last available checkpoint in the .uda file, using restart as the command line argument.

Scheduler: Uintah's scheduler determines the order of tasks and ensures correct inter-processor data is made available. In case there is a dependency that connects two tasks within the same MPI node, it is internal dependency. A dependency that connects tasks across MPI nodes, is an external dependency. Unified scheduler (Figure 5) (uses MPI across nodes and Pthreads on-node), concurrently schedules tasks in a node without a central control thread. Threads, when not executing tasks or waiting for MPI Receive message, continuously monitor two queues - `InternalReadyQueue` and `ExternalReadyQueue`, for tasks with resolved dependencies. If a task's internal dependencies are satisfied, then the task is placed in the internal queue, where it waits for completion of required MPI communication. When the MPI communication is completed, the tasks are moved to the external queue where the task is executed by the first available thread. This particular capability of pushing the task back into queues, enable re-execution of failed tasks during core failures.

2.3 Interpolation

Interpolation is a popular numerical technique that is used frequently to approximate or predict data values from within the given set of values. There are many variations of this technique, varying based on the number of points used and acceptable running times. For a given set of $n+1$ points, a unique polynomial of order n passes through all the points.

$$g(x) = p_0 x^0 + p_1 x^1 + p_2 x^2 + \ldots + p_n x^n \tag{1}$$

To compute the $n+1$ coefficients $p_0, p_1, \ldots, p_n$, $n+1$ unique linear equations are necessary for a unique solution. There are different numerical techniques to solve a system of linear equations, that can be either direct or iterative. For four points, there exists a polynomial of order $4-1=3$, three points give us a polynomial of degree two, and two points give us a polynomial of degree one. The error term between $g(x)$ and the exact value, denoted by a function $f(x)$ can be computed as

$$e_n(x) = f(x) - g(x) = \frac{f^{n+1}(\xi)}{(n+1)!}\psi_n(x) \tag{2}$$

where $\psi_n(x) = \pi_{i=0}^{n}(x - x_i)$. The accuracy of an interpolated point is improved by using a higher order polynomial interpolation technique, which could be expensive to compute as n increases.

Linear interpolation, used between a pair of adjacent points, lacks continuity (C^0) and by extension derivative. Cubic interpolation, which uses 4 points, say (x_0, x_1, x_2, x_3), two each on the either side, makes the polynomial continuous and differentiable at the end points. An unconstrained derivative at x_0, x_3 makes cubic interpolation C^1 continuous and cubic spline intepolation C^2 continuous. In cubic spline, the left derivative at x_3 equals the right derviative at x_3. Similarly, the right derivative should equal the left derivative at x_0. The interpolation we used in this paper, uses a bounded cubic interpolation as in [12] for interpolating the values at the interior patches, a divided difference scheme at the boundary patches.

Linear Interpolation [4] is the easiest to implement with order of accuracy, (see equation 2) of $\mathcal{O}(h^2)$, where h is step size in a particular direction. Higher order polynomials, such as Bernstein interpolation [1], Hermite Interpolation [3] and Shepard interpolations [5] have been studied within the context of Uintah architecture to find a better alternative to linear Interpolation. Most of the higher order polynomials suffered from Runge phenomenon, as in [6] and can be seen in Figure 6.

The current capabilities of the Uintah framework to deal with fail stop errors is by restarting from the last available checkpoint. Reli-

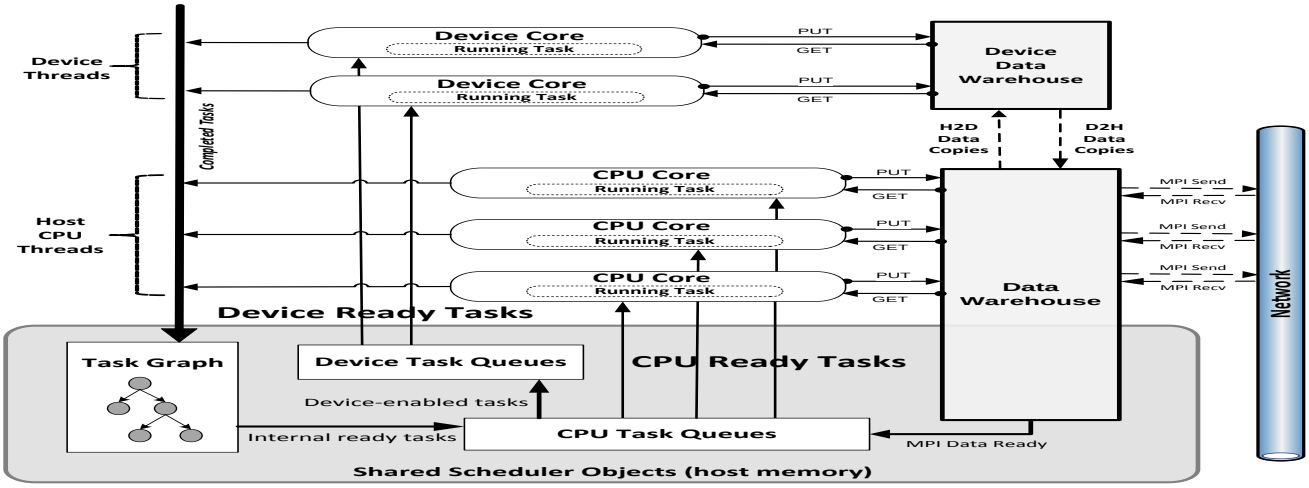

Figure 5: Unified Scheduler in Uintah

ability is a major challenge [2] mentioned by Exascale Task Force on the road-map to Exascale computing. With an uncertainty between the hardware vendors and application developers, on the responsibility to address component failures, we describe techniques of re-execution and interpolation techniques based on [6], in the next sections.

3. CORE FAILURE RESILIENCY

Increases in the number of transistors per core are reaching an end due to the heating effects of Silicon and limitation on the number of transistors within a given area. Most modern architectures use multicore and an on node accelerator to sustain the Moore law. These designs have caused a higher incidence of failures to be reported due to core failures [22] on some of the fastest supercomputers.

Uintah's solution to this problem has been possible by tweaking its hybrid scheduler - Unified Scheduler (Figure 5). In the first step towards building a resilient run time system, we construct a single interface that models system shutdown at the level of a node. This hardware level failure simulation can be done by having randomized behavior in a task that causes it to stop, setting a flag that is equivalent to a system failure response. This simple interface will allowed us to experiment with the run time system and to reschedule task execution by reinserting them into the task queue.

The scheduler concurrently executes tasks in a node without a central control thread. Each *task execution thread* is pinned to a separate core by setting their CPU affinity to core number and thereby avoiding double booking. We model core failure within a node by choosing a random moment within the time-frame of simulation experiment. In case a task is being executed by the thread at the time of failure, we retrieve the task and push it back into the external queue, while also reverting any state related to the reassigned task, e.g. completed task counters, timing statistics, etc. Further executions on the core (in subsequent timesteps) are prevented by not sending a signal to the faulty core's affiliated threads. We performed the same set of experiments on multiple cores by killing them one by one as well as randomly to study the ideal time to declare node failure and trigger the load balancer to reassign patches across other nodes.

4. NODE FAILURE RESILIENCY

Just as a task is lost when a core fails, at least one patch is lost when a node fails. This failure is recovered by storing the coarser version of the patch on a different node and using the interpolation described below as a starting point. We use a grid having 27 patches (3-by-3-by-3 patches per grid) as an example. Consider a cell having 8 node centered variables, we attempt to store just one node variable per cell and try to regenerate the remaining 7 nodal values. We interpolate eight cells (2-by-2-by-2 cells) at a time, with the values stored at (0,0,0), (2,0,0), (0,2,0), (2,2,0), (0,0,2), (0,2,2), (2,0,2) and (2,2,2), labelled from a to h. The goal is to compute the values at cenres of the edges (12 points), centres of the faces (6 points) and the cell centre (1 point). Linear interpolation computes these points, say $u_I^1(x)$ by calculating the average of two nodes for an edge centre, one-fourth of four nodes values for a face centre, and one-eigth eight nodes for a cell centre. As mentioned in 2.3, the accuracy of this technique is $\mathcal{O}(h^2)$. A higher order polynomial interpolation method can be explained using the below four cases, as shown in Figure 7.

Case(a) - Edge centre in an interior patch: We derive the edge case for patch 13 of the grid (the central patch), the face and centroid cases can be extended to two and three dimensions respectively. On the positive x axis, let the indices of (c,a,b,d) be {(-2,0,0), (0,0,0), (2,0,0), (4,0,0)}. The value at x, indexed as (1,0,0);

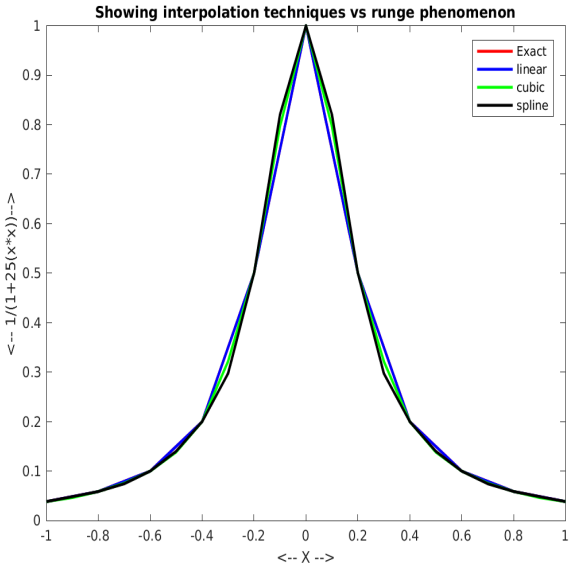

Figure 6: Showing various interpolations and Runge phenomenon

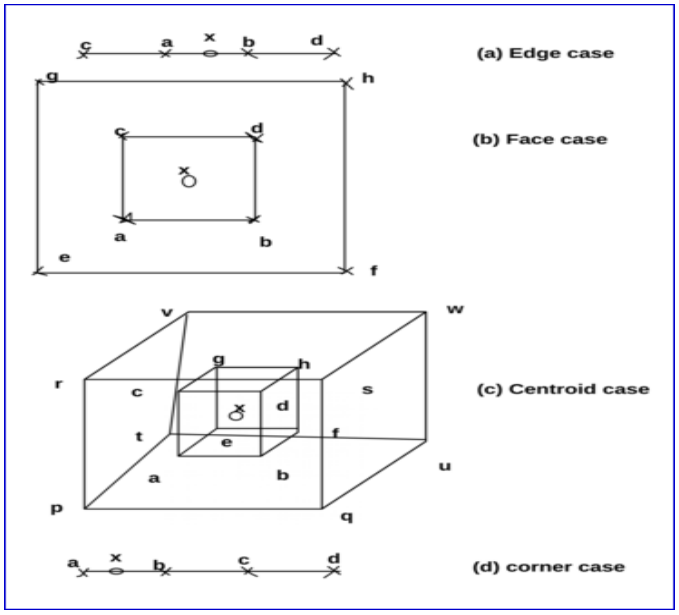

Figure 7: Interpolating cases without dependencies

the midpoint between a and b - represented by a circle, can be computed as

$$u_x = \frac{u_a + u_b}{2} + \frac{\Delta x^2}{2} u_x(x) + \Delta x^4 \tag{3}$$

Similarly, using the values at c and d, u_x can be computed as :

$$u_x = \frac{u_c + u_d}{2} + \frac{9\Delta x^2}{2} u_x(x) + \Delta x^4 \tag{4}$$

Subtracting the latter from the former to eliminate the Δx^2 term and simplifying:

$$u_x = \frac{u_a + u_b}{2} + \frac{1}{8}(\frac{(u_a + u_b)}{2} - \frac{(u_c + u_d)}{2}) + \Delta x^4 \tag{5}$$

To bound the value of u_x between the values of u_a, u_b, we assign the lower and upped bounds based on the smaller and greater of (u_a, u_b) respectively. The goal is to get $min(u_a, u_b) \leq u_x \leq max(u_a, u_b)$. Assuming,

- $\frac{1}{2}(u_a + u_b) = m + (1 - \lambda_1)(M - m), 0 \leq \lambda_1 \leq 1$
- $\frac{1}{2}(u_c + u_d) = m + (1 - \lambda_2)(M - m)$, with no bounds on λ_2

Solving by substituting the values in the equation 5 , we get:

$$u_I^{2*}(x) = m + (1 - \lambda_1)(M - m) + \frac{1}{8}[(1 - \lambda_1) - (1 - \lambda_2)](M - m)$$
$$u_I^{2*}(x) = m + (1 - \frac{9\lambda_1}{8} + \frac{\lambda_2}{8})(M - m)$$
$$u_I^{2*}(x) = m + (1 - \lambda^*)(M - m)$$

where $\lambda^* = MAX(0, MIN(1, {}^{9\lambda_1 - \lambda_2}/_8))$ We repeat this step 12 times for all the edge midpoints, taking the corresponding axis as the direction to interpolate. The case works because, we assumed two layers of ghost cell communication between the patches. When there are less than four cells but two or more cells to the right of node a the direction is reversed, that is we interpolate to the left.

Case(b) - Face centre in an interior patch: For the same patch 13, the linear interpolant value for face centre u_x is :

$$u_I^1(x) = \frac{1}{4}(u_a + u_b + u_c + u_d)$$

We assume that the value of u_x to be within the values of (u_a, u_b, u_c, u_d). To bound the value, we go a level higher. Assuming that a,b,c,d are the nodes on the XY plane with coordinates {(0,0,0), (2,0,0), (0,2,0), (2,2,0) }. To get the bounding face , we go two cell layers deep for the values from nodes (e,f,g,h). Deriving the value at x (1,1,0), similar to the edge case for the six faces as:

$$u_I^2(x) = \frac{u_a + u_b + u_c + u_d}{4} + \frac{1}{8}(\frac{(u_a + u_b + u_c + u_d)}{4}) - \frac{1}{8}(\frac{(u_e + u_f + u_g + u_h)}{4}) + \Delta x^4 \tag{6}$$

The the values of λ_1, λ_2 for this interpolation are

- $m = min(u_a, u_b, u_c, u_d)$
- $M = max(u_a, u_b, u_c, u_d)$
- $\frac{1}{4}(u_a + u_b + u_c + u_d) = m + (1 - \lambda_1)(M - m), 0 \leq \lambda_1 \leq 1$
- $\frac{1}{4}(u_e + u_f + u_g + u_h) = m + (1 - \lambda_2)(M - m)$, no bounds on λ_2

Substituting the values in Equation 6, we have

$$u_I^2(x) = m + (1 - \lambda_1)(M - m) + \frac{1}{8}[(1 - \lambda_1) - (1 - \lambda_2)](M - m)$$
$$u_I^2(x) = m + (1 - \frac{9\lambda_1}{8} + \frac{\lambda_2}{8})(M - m)$$
$$u_I^{2*}(x) = m + (1 - \lambda^*)(M - m)$$

where $\lambda^* = MAX(0, MIN(1, {}^{9\lambda_1 - \lambda_2}/_8))$

Case(c) - Cell centre in an interior patch: We try to compute the value at x with coordinates at (1,1,1). The enclosing 8 nodes, labeled $(a, \ldots, h)$ are given by {(0,0,0), (2,0,0), (0,2,0), (2,2,0), (0,0,2), (2,0,2), (0,2,2) and (2,2,2) }. The bounding cube *PQRSTUVW* is two cell layers outside. Repeating the steps as in case (a) , (b):

$$u_I^1(x) = \frac{u_a + u_b + u_c + u_d + u_e + u_f + u_g + u_h}{8}$$

Bounding values can be solved as

$$u_I^2(x) = \frac{u_a + u_b + u_c + u_d + u_e + u_f + u_g + u_h}{8} + \frac{1}{8}(\frac{(u_a + u_b + u_c + u_d + u_e + u_f + u_g + u_h)}{8}) - \frac{1}{8}(\frac{(u_p + u_q + u_r + u_s + u_t + u_u + u_v + u_w)}{8}) + \Delta x^4 \quad (7)$$

The the values of λ_1, λ_2 for this interpolation are

- $m = min(u_a, u_b, u_c, u_d, u_e, u_f, u_g, u_h)$,
- $M = max(u_a, u_b, u_c, u_d, u_e, u_f, u_g, u_h)$
- $\frac{1}{8}(u_a + u_b + u_c + u_d + u_e + u_f + u_g + u_h) = m + (1 - \lambda_1)(M - m), 0 \leq \lambda_1 \leq 1$
- $\frac{1}{8}(u_p + u_q + u_r + u_s + u_t + u_u + u_v + u_w) = m + (1 - \lambda_2)(M - m)$, no bounds on λ_2

Substituting the values in Equation 7, we have

$$u_I^2(x) = m + (1 - \lambda_1)(M - m) + \frac{1}{8}[(1 - \lambda_1) - (1 - \lambda_2)](M - m)$$

$$u_I^2(x) = m + (1 - \frac{9\lambda_1}{8} + \frac{\lambda_2}{8})(M - m)$$

$$u_I^{2*}(x) = m + (1 - \lambda^*)(M - m)$$

where $\lambda^* = MAX(0, MIN(1, {}^{9\lambda_1 - \lambda_2}/_8))$

Case(d) - Edge centre in a boundary patch: Cases a, b, and c works fine if there are at least two cells in each direction from the interpolating cells. Eight values can be used to regenerate the remaining 19 values of the 8 cells. When a patch is on the boundary, irrespective of the direction, we need a newer scheme. Using a simple Dirichlet boundary condition reduces the accuracy of the interpolated points to linear value. We resolved this issue based on the divided differences scheme of equally spaced meshes described by Berzins in [6]. To have an order of accuracy of Δx^4 we limit ourselves to two divided differences. For edge case, on a positive x axis, coordinates of (a,b,c,d) are {(0,0,0),(2,0,0),(4,0,0) and (6,0,0) }. We compute the value of x located at (1,0,0) with the given scheme.

$$u_x = u_a + (x - a)\frac{(b - a)}{h}(1 + \frac{x - b}{c - a}\lambda_{01}^{12}(1 + \frac{x - c}{d - a}\lambda_{012}^{123}))$$

$$\lambda_{01}^{12} = \frac{u_c - 2u_b + u_a}{u_b - u_a}$$

$$\lambda_{012}^{123} = \frac{u_d - 3 * u_c + 3 * u_b + u_a}{u_c - 2 * u_b + u_a}$$

Simplifying the equation

$$u_x = u_a + 0.5(u_b - u_a)(1 - 0.25\lambda_{01}^{12}(1 - 0.25\lambda_{012}^{123}))$$

$$\lambda_{01}^{12} \notin [-1, 1], \lambda_{01}^{12} = 0$$

$$\lambda_{012}^{123} \notin [-1, 1], \lambda_{012}^{123} = 0$$

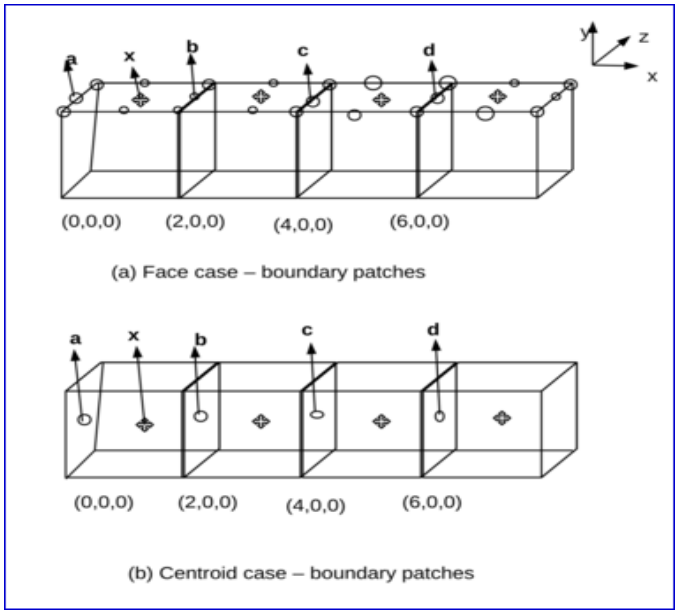

Figure 8: Interpolating at the Boundary patches, having dependencies

The worst case is when :
$\lambda_{01}^{12} = -1, \lambda_{012}^{123} = -1 \implies u_x = u_a + \frac{13}{16}(u_b - u_a)$
similarly
$\lambda_{01}^{12} = 1, \lambda_{012}^{123} = -1 \implies u_x = u_a + \frac{3}{16}(u_b - u_a)$
On all the 12 edges, depending on the direction of the principal axes, we use the same scheme to compute the midpoints. The 2-by-2-by-2 cells currently have 8 nodes and 12 edge midpoints derived. All the cells are either solved with the case(d) or using the earlier cases. To compute the face centre on the boundary patch, we apply the same logic on the midpoints of the edges, instead of the nodes. The cell center can be computed after the face centres are solved in one direction. The figure 8 shows the two cases where the face centres are computed using the edge centres and the cell centres from the face centres.

5. EXPERIMENTAL RESULTS

Based on the techniques described previously, we have tested our code on two of the problems found within the Uintah Examples code base - 3D Poisson solve and the 3D Burgers equation. The Poisson equation uses a 7 point stencil, using the neighboring 6 points (-1,0,0), (1,0,0) (0,-1,0), (0,1,0), (0,0,-1), (0,0,1) to compute the newer value. The number of iterations was set to 100 and each iteration is considered a time step. Figure 9 shows a 2D representation, generated from MATLAB code. In each iteration the value of `phi` is stored and we set the threshold, such that the simulation runs for at least 100 steps.

The second problem, we intend to run our interpolation technique is viscid burger equation. Many interpolation techniques perform well on a smoother domain, but a steep change in values can make the interpolation technique imprecise as in Figure 10. The figure shows a 1D burger code in MATLAB examples, with the diffusion coefficient ν varying in steps of 0.01. The blue line, which is 1 between (-1,0.02) and 0 at the remaining places has a diffusion coefficient of 0.07. That value was used for a lot of our experiments. Retaining positivity in simulations involving chemical concentration is important and we describe a simulation that

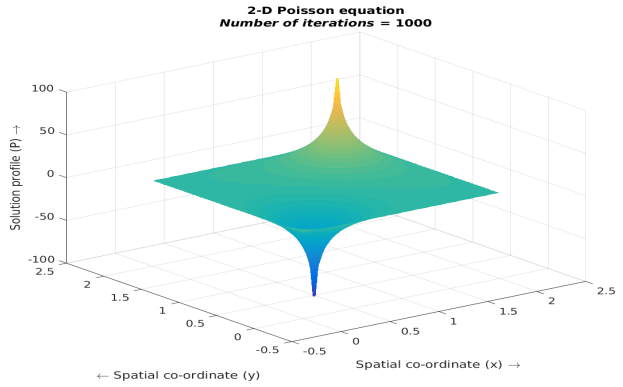

Figure 9: 2D poisson equation

has steep fonts in all three dimensions:

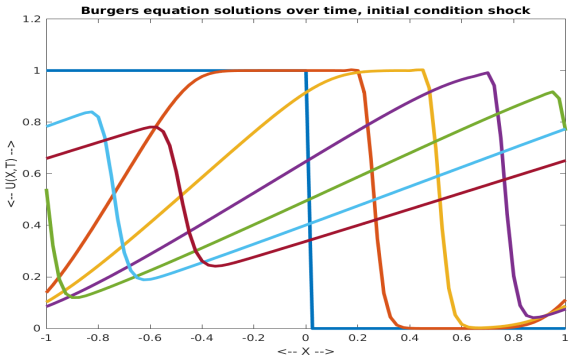

Figure 10: 1D Burger viscid equations over 100 timesteps

$$\frac{\partial u}{\partial t}+w(x,t)\frac{\partial u}{\partial x}+w(y,t)\frac{\partial u}{\partial y}+w(z,t)\frac{\partial u}{\partial z}=\nu(\frac{\partial^2 u}{\partial x^2}+\frac{\partial^2 u}{\partial y^2}+\frac{\partial^2 u}{\partial z^2}) \tag{8}$$

and the analytical solution is

$$u(x,y,z,t)=w(x,t)w(y,t)w(z,t),\mid w(x,t)=\frac{0.1A+0.5B+C}{A+B+C} \tag{9}$$

where $A=e^{-0.05(x-0.5+4.95t)/\nu}$, $B=e^{-0.25(x-0.5+0.75t)/\nu}$ and $C=e^{-0.5(x-0.375t)/\nu}$

All the dimensions are independent and can be represented as

$$\frac{\partial w(x,t)}{\partial t}+w(x,t)\frac{\partial w(x,t)}{\partial x}=\nu\frac{\partial^2 w(x,t)}{\partial x^2} \tag{10}$$

$$\frac{\partial w(y,t)}{\partial t}+w(y,t)\frac{\partial w(y,t)}{\partial y}=\nu\frac{\partial^2 w(y,t)}{\partial y^2} \tag{11}$$

$$\frac{\partial w(z,t)}{\partial t}+w(z,t)\frac{\partial w(z,t)}{\partial z}=\nu\frac{\partial^2 w(z,t)}{\partial z^2} \tag{12}$$

When the right hand side is 0, we have an inviscid burger equation where no diffusion is possible

Both the problems were used to measure running times for a core failure and node failure respectively. We run the set of experiments, mentioned below, to measure the accuaracies of bounded interpolation routine against a standard linear interpolation. In the linear interpolation, the edge centre is the average of the two bounded values, the face centre is average of the four corners and the centroid is one-eighth of all the 8 corners.

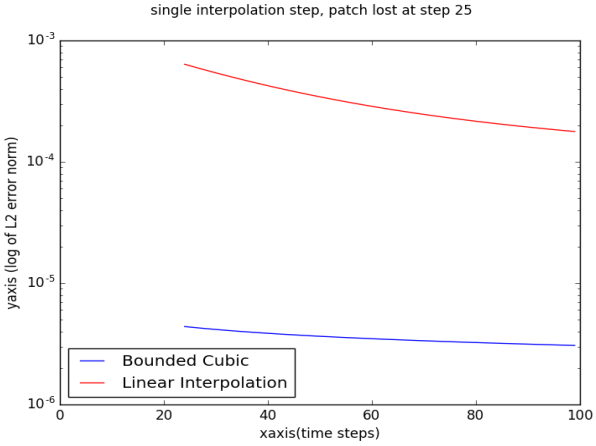

Figure 11: Viscid burger eqn $\nu = 0.07$, Single patch fails at single step 25

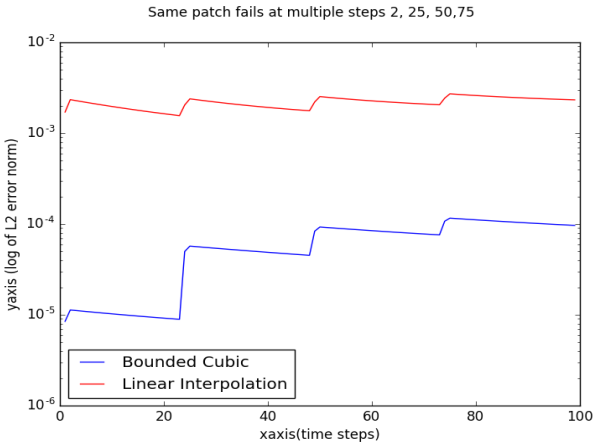

Figure 12: Viscid burger eqn $\nu = 0.07$,One patch keeps failing at multiple steps

- Testing with a single patch failure, at various stages of simulation.
- Single patch fails at a shock location and repeats randomly in the simulation.
- Multiple patches fail within a single step, example of multiple patches assigned to single node.
- Multiple patches fail at varying stages in the simulation.
- Varying the viscosity parameters to interpolate at steep fonts.
- Refining the mesh size to measure the error on a fine mesh vs coarse mesh.

To run a simulation in Uintah, the various parameters of simulation algorithms are set in an input XML file called *.ups file*. To trigger the resiliency routines, we modified the XML schema to add the resiliency components, such as faulting patches, timesteps and

the interpolation technique to be used. Fault detection or monitoring is not necessary with our experiments, as recovery routines are triggered immediately after the timestep is executed. Running an interpolation routine modifies the task graph which has to be rebuilt immediately after the timestep is completed. Modifying the taskgraph is a very expensive process and should be avoided.

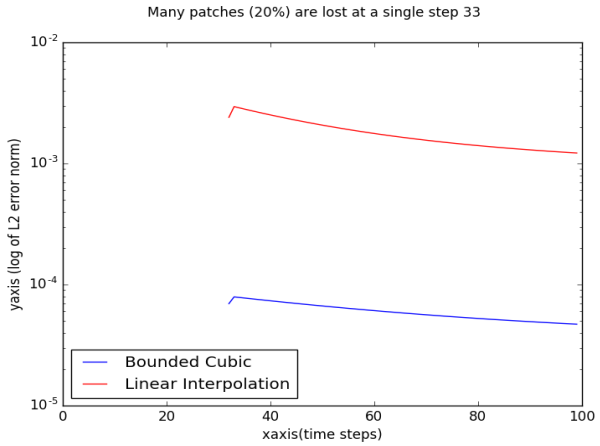

Figure 13: Viscid burger eqn $\nu = 0.07$, Multiple patches fail at a single step

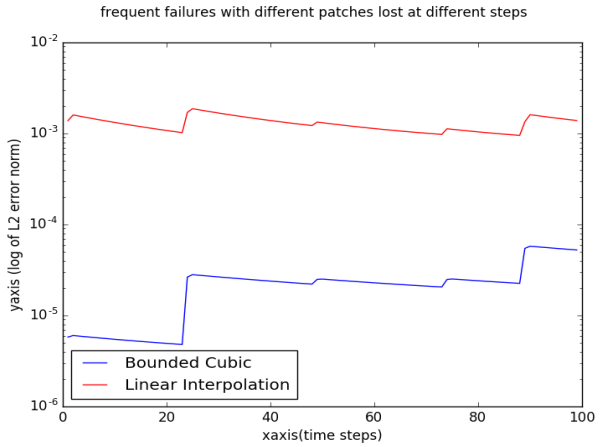

Figure 14: Viscid burger eqn $\nu = 0.07$, random failures, random steps

Node Failures: For our experiments, we have used even number of cells within the failing patch and refinement was increased by a factor of 2 every time. From the results, we see that when a single patch fails, and there were 512 cells within a patch, out of which we use the corner cases on the faces, which are 48 2-by-2-by-2 cubes. The remaining cubes use the bounded interpolation. The L2 error norm for these cases when compared to a failure free simulation was found to be in the range of 10^{-5}. When the diffusion coefficient was tested with $\nu = 0.001$ as in Figure 15, we see the difference in accuracies between linear and cubic bounded was reduced to a factor of 10. When the value of $\nu = 0.07$, as in Figure 11 we see that, the cubic bounded Interpolation has better accuracy, often by a factor of 80 or more. The rate at which the error decays in figures 15 and 11 also indicate that higher the diffusion coefficient the steeper the rate at which the error decays.

In the second set of experiments, we tested to see if there would

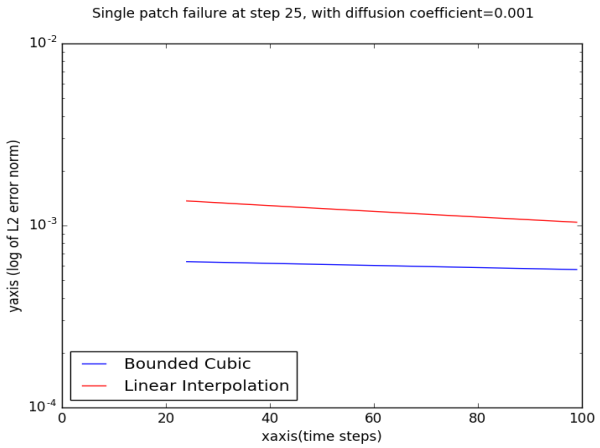

Figure 15: Viscid burger eqn $\nu = 0.001$, Single patch fails at single step

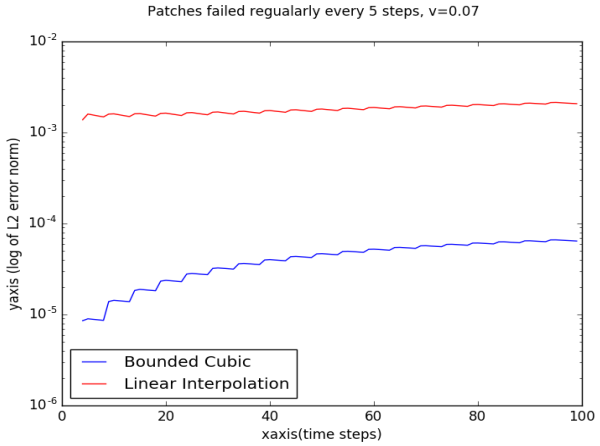

Figure 16: Viscid burger eqn $\nu = 0.07$, periodic failures, every 5 steps

be a variation in the accuracies due to the location of a patch during failure. We tested on a 3-by-3-by-3 computational grid and tested the error norm by failing the patches one by one at the same time step and we observed that patches 3,6,10 gave us a maximum error difference (L_∞) of order 10^{-2} and L-2 of 10^{-4}, see Figure 13. As the failures occur at the patches having a steep font, the overall accuracy of both linear and cubic bound was lower than when the fault was away from the steep font.

Load Balancer in Uintah checks to see if there is a load imbalance by the time a particular processor takes time to execute a task. There is continuous feedback per timestep by loadbalancer, that decides when a re-balance is required. When a node, having multiple patches assigned to it fails, there is a load rebalance necessary. We tested the accuracy with up to 20% of the total patches (6 out of 27) were failed. We see the cubic Interpolation performing better even when recovering from multiple failures, see Figure 13. This figure is an ideal representation of how Uintah behaves during a node failure.

The final set of node failure experiments shows what happens if we have random patch failures throughout the simulation. It was previously assumed that most resiliency techniques would just handle a single failure. We wanted to see how many failures could our

cubic Interpolation handle before the error accuracies becomes as worse as linear. To our surprise, see figures 14 and 16 even when there were failures every other timestep, cubic bound Interpolation performed at least two times better than linear by the end of the timesteps. Such frequent failures were bad for the computation time as the taskgraph recompilation was triggered on every step, to handle interpolation tasks.

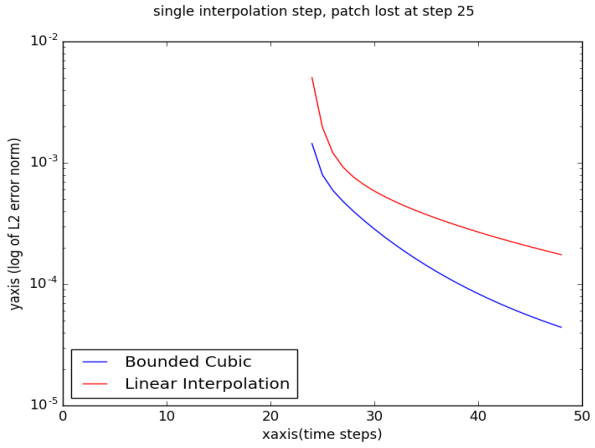

Figure 17: Poisson equation node failure,Single patch fails at single step

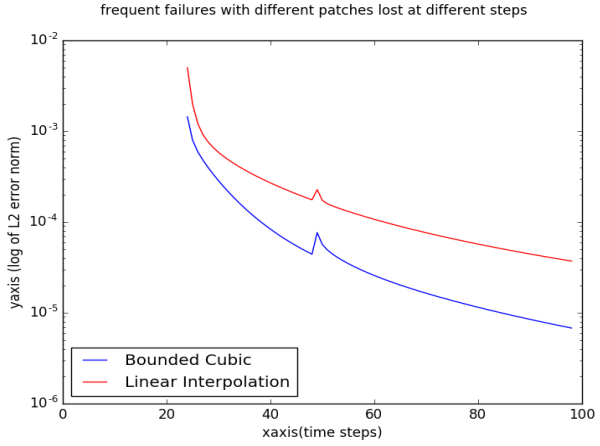

Figure 18: Poisson equation node failure, multi patch at 25, 50, 75 steps

Poisson Equation: Interpolating on Burger equation with a steep font as in Figure 15 was a hard problem but our interpolation technique performed well against linear interpolation. We compared the interpolations with a simple PDE - poisson equation. We observed that the rate of decay of the error was exponential and cubic bound interpolation had better accuracy than linear. The reason the exponent of error was similar for both linear and cubic can be traced to numerical scheme, 7 point stencil for Jacobi iteration, is similar to the interpolation schemes. See figures 17 and 18 which show similar behavior.

While we have not run any large scale scalability experiments, other than running on a 8 core machine with quad-core processor, we have not observed any noticeable difference to the running time to the simulation. This is something, we are interested in studying in future as the memory requirements are not significant in the code. From the table 1, we see that not much of the running times have changed compared to the no fault case. We presume that the impact of adding the interpolation task was limited by the MPI Communication costs.

Table 1: Running times for the number of cells with cubic Interpolation and linear interpolation, averaged 5 runs

Cells per patch	24^3	48^3	72^3	96^3
No Fault (in sec)	10.40	14.37	25.34	47.19
Modified I (in sec)	10.81	14.69	24.77	47.40
Linear I (in sec)	11.40	14.90	25.26	46.34

Mesh sizes vs error norm: In this set of experiments with diffusion coefficient $\nu = 0.07$, we wanted to measure the error norm when the mesh is refined along each direction. We began the experiments with a patch size of 24*24*24 having a ghost layer of 1 cell in each direction, for a total of 15625 (25*25*25) cells per patch. To measure the L2 error norm, we changed the number of cells in a patch to $48^3, 72^3, 96^3$ and the error norms are shown in the Figure 19. While, we couldn't derive meaningful conclusions by measuring the L2 error norms at the end of the simulation, the plots show the initial error before the ghost cells are updated. The logarithmic y axis indicates, the error decays at least by a factor of 2.

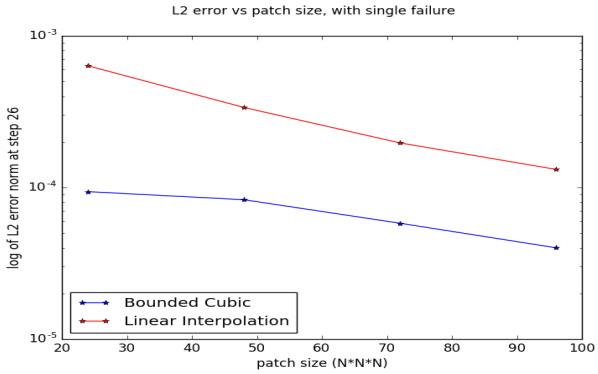

Figure 19: Mesh sizes vs error norm at the failure step

Core Failures: To test the capability of declaring node failure in Uintah due to core failure, we customized the Poisson simulation to retrieve the task executed at the time of core failure and pushing task back to the queue. We tried determining the ideal time to declare node failure by running this simulation for over 100 time steps, with 16 patches on a quad−core dual socket Intel Xeon machine. The mean time to execute a task within a time step was 42 ms. We assumed for this report that, a thread will not be executing a reduction task at the time of failure or that the main thread will not fail. We have altered the input file for parameters such as having a dynamic load balancer and higher accuracy to avoid early completion. A total of 3900 tasks were run over the course of simulation, with times averaged over 5 runs per simulation. Table 2 shows the times for running the simulations ending with no core failures. We measured the elapsed times of simulation that began with 8 threads and having random number of threads fail at random intervals, averaged over 5 runs. The noticeable delay in running times can be seen in Figure 20. It is clear that the impact of a single core failure

can slow down the execution of the node and the entire simulation in general. While the load balancer was not triggered automatically in the simulation, migrating few patches on detection of a failure to load balance might be necessary.

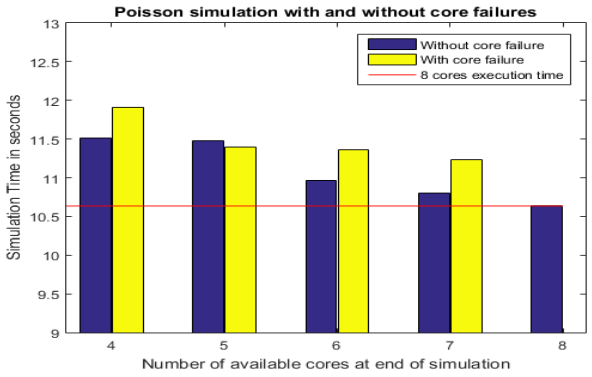

Figure 20: Poisson core failures, multi cores fail per timestep

Table 2: Baseline times for running failure free Poisson simulation having 39 tasks per iteration, for 100 iterations

Thread count	4	5	6	7	8
Time (s)	11.516	11.48	10.959	10.8	10.634

6. CONCLUSION AND FUTURE WORK

In this report, we have discussed techniques to address resiliency concerns when there are core or node failures. While using a checkpoint and recovery technique can be considered an easy way for most of the smaller simulations, this technique will not be feasible for the larger machines. We also present a novel interpolation technique, whose order of accuracy is at least h^4, where h is the step size. From the results from the experiments in section 5 we see that the interpolation we have used in our code performs better than linear interpolation in all the possible cases for the simulations we have tested. We tested the accuracies on the boundaries of the patches and measured the time and measured the number of cores that should fail to declare a node as failure. While extending the cubic bounded Interpolation to other simulations is straight forward, we are yet to implement the code for Cell Centred or Face Centred variables. Apart from these, modifications to input XML schema was done to build resiliency features into the Uintah framework.

Most of the work in this report focused on building resiliency techniques within the Uintah framework. As an application of our interpolation scheme, we intend to extend the work to the prolongation step of AMR. Currently, the techniques to perform prolongation are linear or variations of linear interpolation(Trilinear Interpolation). As demonstrated, we believe using our interpolation technique could improve the accuracies in AMR. We also intend to write a mathematical treatment for the errors introduced in the subsequent steps and how they would influence the overall simulation. An interesting idea would be to characterize the error growth if there are multiple errors within the simulation. During core failure tests, we assumed that cores won't fail during a reduction task or if there is MPI communication. This limitation would be removed in our future work as the majority of the timesteps involve MPI communication. As of now, we have separated the node failure and core failure experiments, but we intend to trigger the recovery routines of node failure when a majority of cores have failed.

7. ACKNOWLEDGMENTS

This material is based upon the work supported by an award received from the National Science Foundation, XPS award number 1337145. The author would like to thank Professor Martin Berzins for the help and guidance provided throughout the course of the project. The author also acknowledges the contributions of Sahithi Chaganti, who studied the results of other higher order polynomial interpolation routines.

8. REFERENCES

[1] Bernstein interpolation. https://en.wikipedia.org/wiki/Bernstein-interpolation.

[2] Exascale challenges. http://science.energy.gov/ /media/ascr/ascac/pdf/reports/Exascale-subcommittee-report.pdf.

[3] Hermite interpolation. https://en.wikipedia.org/wiki/Hermite-interpolation.

[4] Linear interpolation. https://en.wikipedia.org/wiki/Linear-interpolation.

[5] Shepard interpolation. https://en.wikipedia.org/wiki/Inverse-distance-weighting.

[6] Berzins, M. Adaptive polynomial interpolation on evenly spaced meshes. *SIAM Review 49*, 4 (2007), 604–627.

[7] Bosilca, G., Bouteiller, A., Cappello, F., Djilali, S., Fedak, G., Germain, C., Herault, T., Lemarinier, P., Lodygensky, O., Magniette, F., Neri, V., and Selikhov, A. Mpich-v: Toward a scalable fault tolerant mpi for volatile nodes. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, SC '02, IEEE Computer Society Press (Los Alamitos, CA, USA, 2002), 1–18.

[8] Bronevetsky, G., Marques, D., Pingali, K., and Stodghill, P. Automated application-level checkpointing of mpi programs. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, ACM (New York, NY, USA, 2003), 84–94.

[9] Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., and Snir, M. Toward exascale resilience. *International Journal of High Performance Computing Applications* (2009).

[10] Chen, Z. Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, IEEE (2008), 1–8.

[11] Chen, Z., and Dongarra, J. Highly scalable self-healing algorithms for high performance scientific computing. *IEEE Transactions on Computers 58*, 11 (2009), 1512–1524.

[12] de Boor, C. On bounding spline interpolation. *Journal of Approximation Theory 14*, 3 (1975), 191 – 203.

[13] Dubey, A., Mohapatra, P., and Weide, K. Fault tolerance using lower fidelity data in adaptive mesh applications. In *Proceedings of the 3rd Workshop on Fault-tolerance for HPC at Extreme Scale*, FTXS '13, ACM (New York, NY, USA, 2013), 3–10.

[14] Elnozahy, E. N., Alvisi, L., Wang, Y.-M., and Johnson, D. B. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys (CSUR) 34*, 3 (2002), 375–408.

[15] Luitjens, J., and Berzins, M. Improving the performance of uintah: A large-scale adaptive meshing computational framework. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, IEEE (2010), 1–10.

[16] Luitjens, J., and Berzins, M. Scalable parallel regridding algorithms for block-structured adaptive mesh refinement. *Concurrency and Computation: Practice and Experience 23*, 13 (2011), 1522–1537.

[17] Meng, Q., and Berzins, M. Scalable large-scale fluid-structure interaction solvers in the uintah framework via hybrid task-based parallelism algorithms. *Concurrency and Computation: Practice and Experience 26*, 7 (May 2014), 1388–1407.

[18] Meng, Q., Humphrey, A., Schmidt, J., and Berzins, M. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, IEEE (2013), 1–12.

[19] Moody, A., Bronevetsky, G., Mohror, K., and Supinski, B. R. d. 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*, SC '10, IEEE Computer Society (Washington, DC, USA, 2010), 1–11.

[20] Peterson, B., Xiao, N., Holmen, J., Chaganti, S., Pakki, A., Schmidt, J., Sunderland, D., Humphrey, A., and Berzins, M. Developing uintahś runtime system for forthcoming architectures. Tech. rep., SCI Institute, 2015.

[21] Stellner, G. Cocheck: Checkpointing and process migration for mpi. In *Parallel Processing Symposium, 1996., Proceedings of IPPS'96, The 10th International*, IEEE (1996), 526–531.

[22] Tiwari, D., Gupta, S., Gallarno, G., Rogers, J., and Maxwell, D. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, ACM (New York, NY, USA, 2015), 38:1–38:12.

[23] Wang, C., Mueller, F., Engelmann, C., and Scott, S. L. A job pause service under lam/mpi+ blcr for transparent fault tolerance. In *2007 IEEE International Parallel and Distributed Processing Symposium*, IEEE (2007), 1–10.