Developing Uintah's Runtime System For Forthcoming Architectures

Brad Peterson[†], Nan Xiao[†], John Holmen[†], Sahithi Chaganti[†], Aditya Pakki[†],
John Schmidt[†], Dan Sunderland[‡], Alan Humphrey[†], Martin Berzins[†]
[†]Scientific Computing and Imaging Institute, University of Utah, Salt Lake City, UT, 84112, USA
[‡]Sandia National Laboratories, PO Box 5800 / MS 1418, Albuquerque, NM, 87185, USA
mb@sci.utah.edu

ABSTRACT

The need to solve ever-more complex science and engineering problems on computer architectures that are rapidly evolving poses considerable challenges for modern simulation frameworks and packages. While the decomposition of software into a programming model that generates a task-graph for execution by a runtime system make portability and performance possible, it also imposes considerable demands on that runtime system. The challenge of meeting those demands for near-term and longer term systems in the context of the Uintah framework is considered in key areas such as support for data structures, tasks on heterogeneous architectures, performance portability, power management and designing for resilience by the use of replication based upon adaptive mesh refinement processes is addressed. Examples of techniques in these areas are used to illustrate performance improvements obtained for present large-scale systems and the needs of proposed systems considered.

Keywords

Uintah, scalability, parallel, adaptive

1. INTRODUCTION

The need to solve larger and more challenging multi-scale and multi-physics science and engineering problems on computer architectures that are rapidly evolving to address improvements in performance and power consumption poses considerable challenges for modern simulation frameworks and packages. A fundamental decomposition in such software is to use a programming model to specify a task form of the application and its associated numerical methods and then to execute these tasks by means of an adaptive runtime system. The potential value of this approach is stated by [20, 30] *Exascale programming will require prioritization of critical-path and non-critical path tasks, adaptive directed acyclic graph scheduling of critical-path tasks, and adaptive rebalancing of all tasks with the freedom of not putting the rebalancing of non-critical tasks on the path itself.* This is the approach used in the Uintah software e.g. [11,43]. The separation, of user code and runtime system and also the runtime system that is used on each compute node, permits us to leverage advances in the runtime system, such as scalability, to be immediately applied to applications without any additional work by the component developer. The nodal component of the runtime system has an execution layer that runs on each core that also queries the nodal data structures in order to find tasks to execute and works with a single data warehouse per multi-core node to access local variables and and non-local variables through MPI communications. Each mesh patch that is executed on a node uses a local task graph that is composed of the algorithmic steps (tasks) that are stored along with various queues that determine which task is ready to run. Data management including the movement of data between nodes along with the actual storage of data in the Data Warehouse occurs on a per node basis. The actual execution of the various tasks are distributed on a per core level. A strength of this approach is that the applications code is portable and indeed may oblivious to changes in architectures. Indeed with this approach Uintah has been able to compute solutions to complex engineering problems on a number of the present most powerful machines such as Titan, Mira, Vulcan, Stampede and Blue Waters, for the complex engineering problems described in Section 2.

An implication of this approach is that the runtime system must constantly evolve [14] to meet the challenges of future and present architectures. The trends over the next five years seem to be that present CPU architectures will continue at small to medium scale, albeit with more power-efficient cores, while many newer scale large architecture will embrace presently-evolving NVIDIA GPUs or Intel Xeon Phi processors. Examples of such machines are the IBM NVIDIA Summit machine with 3,500 nodes and multiple GPUs per node at Oak Ridge National Laboratory and the Intel-Cray Aurora machine with 50,000 Intel Xeon Phi chips at Argonne National Lab. Public overviews of these machines are given at `http://science.energy.gov/ascr/ascac/meetings`.

A key need for the present and future architectures is to address the efficiency of data structures. An example of this is given in Section 3, in which the data structures that support task-graph compilation are revised to improve performance. At the same time heterogeneous architectures with multiple accelerators per node make it important to create the infrastructure to support delay free task execution, of tasks that may have quite short execution times, as in Section 4. The need to combine efficient task execution with portability may be resolved in a number of different ways. One is through the use of domain specific languages such as the Nebo approach used with Uintah [21]. In Section 5 of this paper an alternative approach using the Kokkos intermediate layer is explored [23], as Kokkos makes it possible for core loops to be written in a form that maps the underlying data structures in the most efficient way

onto different architectures. The final challenge to address is that of resiliency, as the runtime system has to evolve to meet the expected challenge of resiliency, by using fault tolerant approaches such as, for example, the fault-tolerant message passing approach of redMPI, [24]. For example If cores automatically reduce clock speeds, then it will be possible to use our present approach to load balance the calculation. If a core or node is not performing or is consuming too much power, then tasks can be moved elsewhere or delayed. In the case of node or core failures we are simulating the approach of using system monitors for hard fault-tolerance. Should calculations involving Uintah patches fail, it is always possible to use checkpoint restarts. It is often also remarked upon that this approach is too expensive if failures are very frequent. In this case in Section 7 we investigate using task replication at a coarse level using Adaptive Mesh Refinement. A task may be replicated on a coarser mesh at 1/8 of the cost of its parent. If the parent task fails then interpolation is used to reconstruct the parent task data. A key aspect of this process is the choice of the interpolation routines which must preserve the physical characteristics of the underlying solution, such as positivity. For this reason a modified Shepard interpolant is proposed and its performance investigated as part of our move to a runtime system that will eventually address all the above challenges of forthcoming computer architectures.

2. UINTAH FRAMEWORK

The Uintah open-source software framework makes it possible to solve a very broad problem class of fluid-structure interaction problems from a variety of science and engineering disciplines, [10]. The central idea [11, 43] is to use a layered approach, see Figure 1, that structure applications drivers and applications packages as a Directed Acyclic Graph (DAG) of computational tasks, belonging to Uintah components that access local and global data from a *data warehouse* that is part of an MPI process and that deals with the details of communication. A runtime system manages the asynchronous and out-of-order (where appropriate) execution of these tasks and addresses the complexities of (global) MPI and (per node) thread based communication. Many Uintah problems are currently

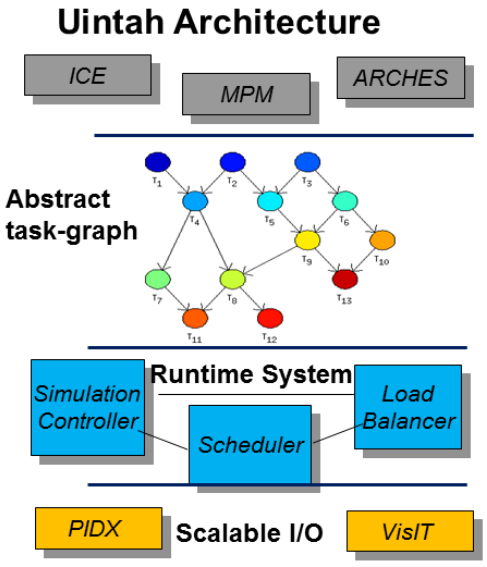

Figure 1: Uintah Architecture

solved by making use of four primary components [11, 43]: 1) a low and high-speed compressible flow solver, ICE; 2) a material point method algorithm, MPM for structural mechanics; 3) a fluid-structure interaction (FSI) algorithm, MPMICE which combines the ICE and MPM components and 4) a turbulent reacting CFD component, ARCHES designed for simulation of turbulent reacting flows with participating media radiation. Each component is expressed as a sequence of tasks in which data dependencies (inputs and outputs) are explicitly specified. These tasks are then compiled into a task-graph representation (Directed Acyclic Graph) to express the parallel computation along with the underlying data dependencies. The smallest unit of parallel work is a patch composed of a hexahedral cube of grid cells. Each task has a C++ method for the actual computation and each component specifies a list of tasks to be performed and the data dependencies between them [13]. Uintah's runtime system executes these tasks independently of the application itself thus allowing the tasks and the infrastructure to be developed separately with scalability concerns such as load balancing, task (component) scheduling, communications, including accelerator or co-processor interaction. Scalability is achieved by continuously improving the runtime system to include features such as scalable adaptive mesh refinement [34] and a novel load balancing approach [33]. While Uintah uses a Directed Acyclic Graph approach for task scheduling, the use of dynamic/out-of-order task execution is important in improving scalability [43]. For systems with reduced memory per core, only one MPI process and only one data warehouse per node are used. Threads are used for task execution on individual cores. This has made it possible to reduce memory use by an order of magnitude and led to better scalability [41]. Additional details surrounding Uintah's runtime system can be found in [43]. Even thought this approach is very successful on a range of existing large machines [43] it is still necessary for the applications user to ensure that there are enough patches per core to hide communications costs, or at least to make them scalable. In going to the next generation of machines the main lesson from previous increases in the scale of Uintah is that while the applications code may stay unchanged both the algorithms used in the runtime system and the runtime system itself have to evolve to take into account the features of the target machines, as will be explored below.

3. DATA STRUCTURES FOR PETASCALE AND BEYOND

For almost any machine that is capable of 100 PF or more it will be necessary to have local data structures that can be easily and efficiently accessed. The Uintah task graph needs to be compiled at the beginning of a simulation or whenever the patch layout changes (recompiled) which typically occurs during an AMR simulation, i.e. patches are refined or coarsened. Task graphs compilation has been shown to be the predominant scalability concern [40] as we approach the multi-million core regime. To address the scalability bottlenecks, several non-incremental algorithms and data structures were implemented. In Uintah, problem domain is decomposed into many patches and each core has a global view of all of these patches. Each core needs to know which patches are inside a given index range from time to time. The compilation phase includes two main steps: finding neighboring patches with their hosts and creating dependencies for tasks inside local patches. The Bounding Value Hierarchy (BVH) tree was the predominant data structure identified which dominated the overall time cost for finding neighboring patches within a given index range. Apart from redesign of these data structures, interfaces between functions and redundant sorting processes were also optimized. The incorporation of

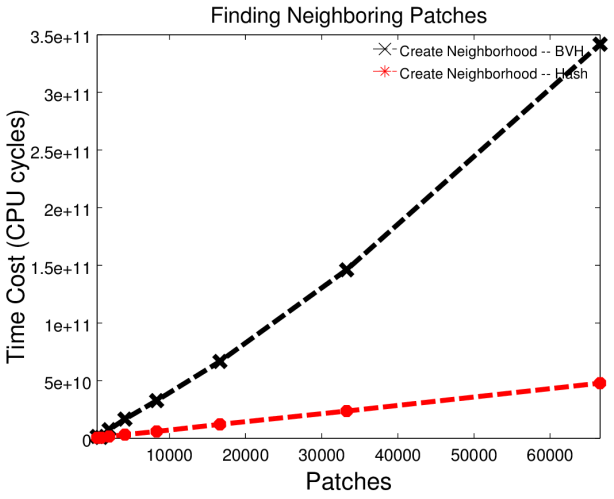

Figure 2: Time to Find Neighboring Patches

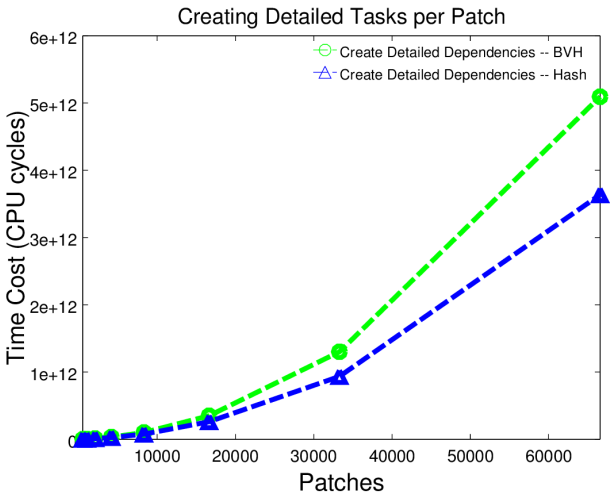

Figure 3: Time to Create Detailed Dependencies

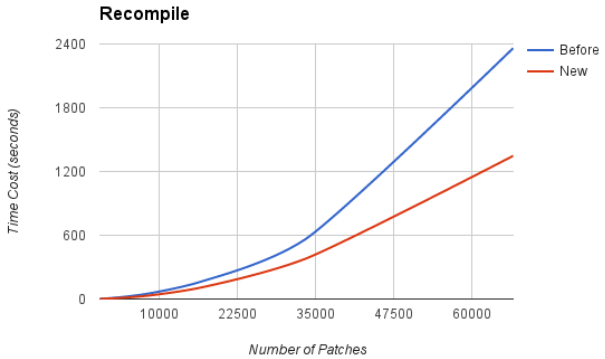

Figure 4: Overall Recompile Time

the new data structures and algorithms reduced the time to perform task graph compilation by approximately 45%, see Figure 4.

Before optimization, BVH tree structure is used for querying patches in the grid during compilation. The time cost for creating the BVH tree is $O(n(\log n)^2)$ and $O(k \log n)$ for each query, where k represents number of patches returned. As part of the creation of the overall Taskgraph such as the dependency creation for the tasks, patches must be sorted. The BVH tree stores the patches randomly requiring an extra sorting process.

Uintah's domain is decomposed into a collection of patches that are geometrically uniformly distributed for each level of the grid hierarchy. We project patches into a continuous space. The IDs of patches are generally assigned based on patch's index and are in increasing order, which usually does not require an additional sort for the returned patches. The query function returns all the matched patches via a hash tree. The hash tree can be thought of as a set of hash tables stored in a heap tree where each node is linked to the head of a hash table. The return order inside each hash table is sorted to accommodate the rare cases when several patches at a finer levels may not be continuous with others leading to a significant increase in memory.

The constructor for this data structure includes domain partitioning, a hashing function, and indexing. The partition algorithm is used to cluster patches into several subdomains based on whether they are adjacent to each other to improve the memory utilization rate. Each subdomain has its own hash table. An indexing method is used to improve the query performance by ensuring each empty bucket points to the next non-empty one. A heap tree combines all hash tables in subdomains. This approach results in improved scalability results for both components of new task graph compilation: finding neighboring patches, and creating tasks dependencies. In order to obtain scaling results shown below, we tested and chose the optimal patch configuration to demonstrate the scalability up to 256 cores on a modest sized HPC machine. Figure 2 demonstrates the overall weak scaling for both old and new functions of finding neighboring patches, Figure 3 demonstrates the overall weak scaling for function creating tasks dependencies, and Figure 4 represents the weak scaling for overall task graph compilation, where we observed approximate 1.8X speedup when running with 256 cores.

4. SUPPORTING HETEROGENEOUS ARCHITECTURES AT RUNTIME

As computer node configurations are becoming more complex, the need for a robust runtime system separate from an application layer also increases. The Uintah framework is built with an assumption that the application layer developer should not need to worry about the details of memory management such as allocating and pinning memory, copying data from one region to another, gathering halo cells (hereafter referred to as ghost cells), etc. Instead the runtime system should intelligently and efficiently prepare all variable data prior to performing any needed computation. This section describes the initial design of Uintah's runtime system for using multiple GPUs per node, describes its flaws, and describes design changes to support a wider range of tasks. The design changes are analyzed to demonstrate flexibility for future node accelerator and memory configurations.

As outlined in [29], the Uintah runtime system was extended to support multiple GPU devices per node for certain problems. This design was kept relatively simple. Prior to a task's execution, all necessary data from the prior time step is copied into a GPU's memory. That task is executed on that GPU. Then computed variables

are copied back to host memory. GPUs are assigned in a round-robin fashion. This approach supported a small set of computations, such as computing radiation transfer using a reverse Monte Carlo ray tracing (RMCRT) algorithm [28]. Each RMCRT task required one second or more to compute, and the time to copy all data in and out of the GPU is a minimal fraction of the computation time. By employing multiple GPUs, a node could be assigned more patches. By copying all computed data back to the host after each time step, the existing runtime system could manage host memory as if the task never ran on the GPU at all, and it would follow the ghost cell scenarios as outlined in Figure 5.

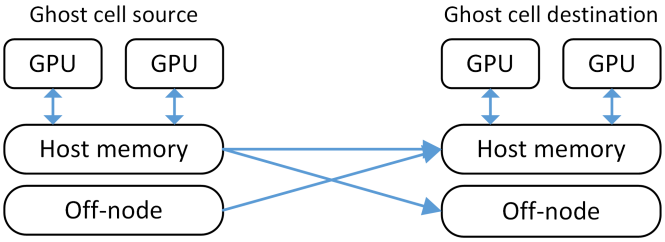

Figure 5: Ghost cell management under the initial runtime engine. A data management task could manage up to three source to destination scenarios, namely, host to host, host to off-node, and off-node to host.

This approach however does not work for many types of GPU tasks. Specifically, GPU tasks which computed within a few milliseconds do not perform well relative to CPU computation times as the overhead to prepare data in the GPU was far greater than the time spent computing the task. Also, GPU tasks which required dozens of data variables suffered from the overhead of API call latencies. The major motivation for this work is to support these GPU tasks and simulations in a multi-GPU environment.

In a general sense, the Uintah runtime system needs to support any number of cores per node, zero or more accelerator devices per node, one or more patches per node, one or more patches per accelerator device, patches partitioned in up to three dimensions, any number of ghost cells per data variable, and this must all operate on any number of MPI ranks. Data variables need to be kept resident on accelerator devices, avoiding unnecessary movement from one memory location into another. The overhead for all memory management should be kept within a few milliseconds or less. And the Uintah runtime system needs to be prepared for any future additional memory hierarchies.

Modifying the runtime system to handle these needs has proven to be challenging. Many recently implemented improvements to support a single GPU per physical node have been outlined in [45], driving down overhead time between time steps to between one and two milliseconds. This section will focus on the design decisions to better support multiple on-node devices.

To support multiple GPUs, it is tempting give each GPU its own MPI rank. If this model was used, data communication between multiple GPUs can be managed simply through CUDA-aware MPI. However, this comes at a cost of increasing the amount of MPI messages, duplicated shared data, and limited work stealing due to each rank having fewer tasks. Uintah is designed that within an MPI rank, Pthreads and shared data warehouses are employed to minimize MPI ranks and accompanying MPI messages [42]. Therefore multiple GPUs are designed to work within a single MPI rank, while allowing for flexibility so that a physical node could be assigned multiple MPI ranks if desired. The resulting model of the current runtime system is shown in Figure 6.

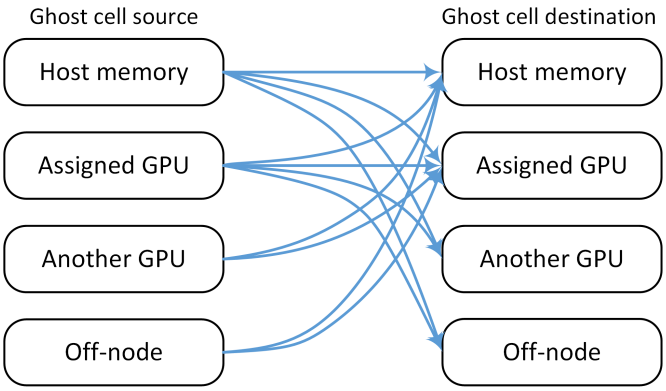

Figure 6: Ghost cell management under the new runtime engine. A data management task is assigned a region of patches and its associated GPU. It could manage up to twelve source to destination scenarios if an MPI Rank contains two or more GPUs.

Currently at the start of a new simulation time step, data management tasks are launched whose job is to identify and manage the data dependencies for upcoming tasks in that time step. For example, if a node has 26 MPI rank neighbors, then 26 data management tasks are created, each one sending MPI messages to its assigned neighbor. Each of these tasks analyzes its own node's task graph and is able to identify all external data dependencies associated with other MPI ranks. To support multiple GPUs in the same MPI rank, this same analysis can be done to find all internal dependencies. If a node has three GPUs, then three additional data management tasks should be created to prepare and process peer to peer copies from its assigned GPU to any other on-node GPU.

Not shown in Figure 6 is the bulk nature in which ghost cells are processed. The number of GPU API memory allocations, number of memory copies, bytes of data copied, and kernel calls should be kept to a minimum. To support this, the runtime system first identifies all allocations and copies that will need to be made prior to performing these actions. They are then processed in bulk. For example, suppose a data management task is assigned a patch containing a data variable in GPU #0. The runtime recognizes that one neighbor patch is in on-node GPU #1. Another neighbor patch is in on-node GPU #2. And another neighbor patch is in another MPI rank. Contiguous staging arrays would be created on GPU #0 for each ghost cell region that needs to sent outside that GPU. A kernel is launched which copies all ghost cell data into contiguous arrays within GPU #0. Contiguous staging arrays would be created in GPU #1 and #2. For the on-node GPUs, peer to peer GPU copies are invoked. For the off-node neighbor, CUDA-aware MPI can be invoked. This completes the duty of that data management task. Later the runtime will analyze computational tasks prior to execution and gather ghost cells from these staging arrays back into their data variables.

From a global perspective, the amount of MPI ranks doesn't change, and the amount of MPI messages do not change. As 6 demonstrates, an MPI rank will will still send messages from from either host memory or GPU memory. The receiving ranks will receive messages and process them accordingly.

Overall this runtime system model is better able to scale as more GPUs are assigned per node. It meets the needed goals of extending Uintah to support the types of GPU tasks and computational

problems that are not realistic or possible on the original Uintah GPU runtime system. This approach works for additional accelerators and memory hierarchies, including Intel Xeon Phis and a tiered hierarchy of host memory. Each additional memory space simply needs its own pool of contiguous staging arrays. For example, suppose host memory was split into two tiers. Then in Figure 6, one additional source and destination region would be managed. Much of this framework has recently been implemented for one GPU per node per MPI rank, and work is progressing for multiple GPUs per MPI rank.

5. PERFORMANCE PORTABILITY THROUGH MAPPING ABSTRACTIONS

With its emphasis on large-scale simulations, Uintah must be able to leverage the increasing adoption of accelerators and coprocessors within current and emerging heterogeneous supercomputer architectures. However, this poses a challenge due to the differing programming efforts required to achieve performance across architectures. To alleviate code bifurcation, efforts are underway exploring the use of Kokkos [23] to help enable performance portability. Here, performance portability refers to the maximization of portability across varying architectures while striving to achieve performance comparable to hand-tuned codes.

Developed by Sandia National Laboratories, Kokkos is a C++ library that allows developers to write portable, thread-scalable code optimized for CPU, GPU, or MIC-based architectures. Specifically, this library provides developers with architecture-aware parallel algorithms (*parallel_for*, *parallel_reduce*, and *parallel_scan*) and data structures. When using these tools, Kokkos is able to select a memory layout and iteration scheme for a target architecture at compile time. This is facilitated through a number of back-ends, which currently support CUDA, OpenMP, and PThreads programming models for NVIDIA GPU, Intel Xeon Phi, and CPU-based machines. To manage thread placement, Kokkos also provides support for the Portable Hardware Locality (HWLOC) [2] package. Detailed usage information and demonstrations of performance portability can be found at the Kokkos tutorial webpage [22].

To best leverage Kokkos, developers are encouraged to use provided data structures and encapsulate algorithm kernels within C++ functors or lambdas to enable use of Kokkos parallel algorithms. To begin our integration of Kokkos, we have enabled the use of Kokkos parallel algorithms within Uintah. Traditionally, Uintah creates task execution threads that are mapped to PThreads on a 1:1 basis for single-threaded task execution. To accommodate Kokkos, we have enabled mapping of Uintah task execution threads to disjoint Kokkos thread pools. This one-to-many mapping allows for use of Kokkos parallel algorithms and multi-threaded task execution. Note however, further modification of the runtime system is required to fully support Kokkos (e.g. modifying the data warehouse to return Kokkos views).

With these changes in place, Figure 7 demonstrates how to transform a Uintah task into one supporting Kokkos. Generally, Uintah tasks feature a header that declares and initializes mesh patch variables. These variables are then used within one or more loops when executing tasks corresponding to a given mesh patch. To support Kokkos, these loop bodies must be encapsulated within C++ functors or lambdas as demonstrated in Figure 7.

As a proof of concept, a Poisson solver was modified to use a Kokkos parallel loop. This example was chosen as the primary work relied on one simple task. After creating a functor and encapsulating the original loop body within a Kokkos parallel loop, experiments were run across two Xeon E5-2680 processors. An

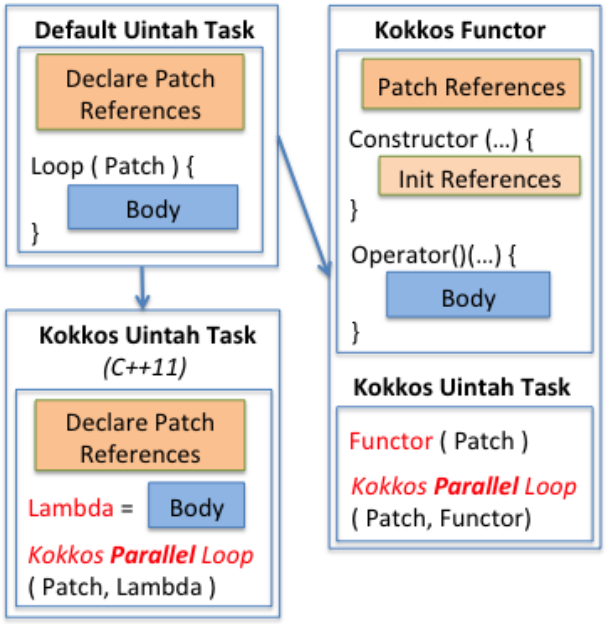

Figure 7: Enabling Kokkos by transforming a standard Uintah task. We can create a functor which has patch variable references as members with a loop body implemented in the parentheses operator, or we can use a lambda which captures the loop body.

Domain Sizes	128^3	256^3	512^3	768^3	1024^3
Grind – Original	1.521	1.978	1.651	1.602	1.550
Grind – Kokkos	0.723	0.840	0.833	0.780	0.847

Table 1: Per cell execution times as a function of the number of cells within a domain.

overview of results from these experiments can be found in Figure 8, with exact results reported in Table 1.

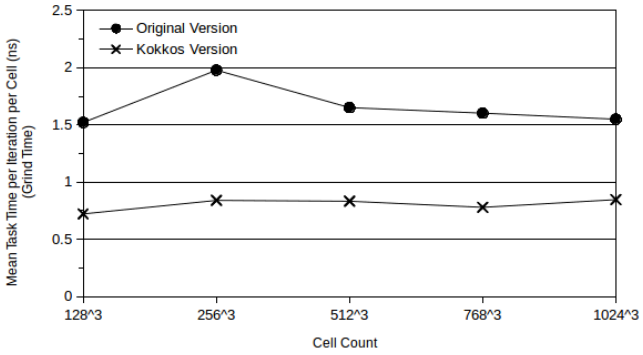

Figure 8: Weak scaling of the Poisson task execution grind time on a Sandy Bridge-based system.

For this proof of concept, emphasis was placed on understanding how use of Kokkos parallel loops impacted task performance. In addition to experimenting with multiple domain sizes, varying thread pool configurations were also explored. Timings depicted in

Figure 8 and Table 1 correspond to the mean task execution time for a single iteration of the Poisson solve on a per-cell basis. Further, these timings correspond to the most advantageous thread pool configuration identified for a given domain size.

These results demonstrate great weak scaling of the per-cell task execution time with increasingly larger domains. A key takeaway is that use of Kokkos parallel loops offered performance improvements across the board. This is likely attributed to their enabling of multi-threaded task execution, which provides greater control over simulation mesh decomposition. For these experiments, use of Kokkos thread pools allowed both hyper-threads within a given core to cooperatively execute tasks. Such results suggest that Kokkos will help better prepare Uintah for running well on current and emerging architectures featuring increasingly larger core counts.

This initial effort has helped increase our understanding of what Kokkos has to offer before pursuing wider-spread migration. In addition to previously mentioned advantages, Kokkos also presents an opportunity to reduce the gap between development time and our ability to run on newly introduced machines. These advantages in mind, Kokkos is believed to be a critical aspect in preparing Uintah for future machines. Next steps presently underway include the integration of Kokkos parallel loops within the Arches combustion simulation component. Looking further ahead, considerations must also be made for incorporating use of Kokkos data structures within Uintah.

6. ADDRESSING RESILIENCE FOR NODE AND CORE LEVEL

The next generation of clustered machines, capable of delivering 100−plus Petaflops with billion way parallelism, are not only expected to have higher number of components but also have a lower Mean Time Between Failure (MTBF). The Blue Waters system, an example of clustered Petascale machine with $362,240$ cores, built from commodity components required remedial action for either hardware or software failures, approximately every 4.2 hours [37] during pre-production stage. As improvements in hardware to withstand failures been slower than the growth in computational power, providing resiliency with the help of software is crucial. We attempt to build a version of Uintah capable of withstanding failures at the level of cores/ accelerators, on-node memory, node, and MPI failure by injecting faults to the simulations through an interface and resolving them using software and existing hardware capabilities. Currently, Uintah uses a checkpoint restart mechanism to deal with fail stop errors with an option to vary the frequency of checkpointing to suit the simulation. At exascale, simple checkpointing is not a feasible approach as low MTBF would require frequent checkpointing and the huge volume of data generated per checkpoint could inhibit forward progress of the simulation.

Our planned approach to making Uintah resilient can be broadly classified into areas involving prediction, detection, notification and correction of hardware and software errors. It is difficult to predict the support for error detection that would be available at the operating system level. Good examples of proposed system layers are given by Figure 9. Examples of monitoring approaches for hardware failures are systems such as the Intelligent Platform Management Interface (IPMI) which can also be used [47] as part of a process to monitor hardware failures. We have attempted to integrate features from Ganglia Monitoring System [38], a scalable grid monitoring system with notification capabilities, to Uintah's run time system but found them to provide poor support for co-processor and accelerator core monitoring. However, low memory allocation overhead per node and hierarchical data structures of Ganglia are ideal characteristics to incorporate in future monitoring and notification systems at exascale.

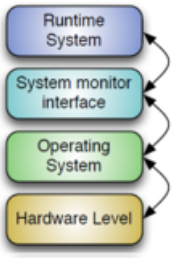

Figure 9: Proposed model of system layers

Silent data corruption (SDC) or software faults affect the data integrity in the system but are harder to detect as a bit flip won't cause the system to fail−stop but could give incorrect results. Using redundancy based RedMPI [24], we compare the messages of a MPI command, executed redundantly, typically twice or thrice, on the sender side and detect a SDC on the receiver side, if the hash code of the message content differs. The results in [24] suggest a trade off in terms of redundant computing might be necessary for detecting and recovering from SDC errors. We intend to use RedMPI with available optimization to minimize communication overheads and maximize the performance to power ratio.

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, with setting a flag that is equivalent to a system failure response. This simple interface will allow us to experiment with the run time system and to reschedule task execution by reinserting them into the task queue. This facility necessitates a change to the scheduler functionality, described below.

The Unified Scheduler, Uintah's hybrid scheduler (using MPI across nodes and Pthreads on-node) 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. Threads, when not executing tasks or waiting for MPI receive message, continuously monitor `InternalReadyQueue` and `ExternalReadyQueue`, the queues having 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. We model the 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 and randomly to study the ideal time to declare node failure and trigger the load balancer to reassign patches across other nodes.

To test the capability of declaring node failure in Uintah, 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 paper 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. The core failure results shown in Figure 10 measure the elapsed times of simulation that began with 8 threads and having random number of threads fail at random intervals, averaged over 5 runs. 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.

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

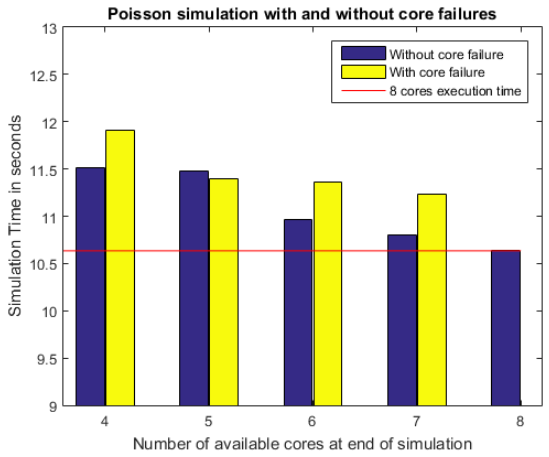

Figure 10: Timing the Poisson equation simulation with and without failure

6.1 Resilience through AMR Replication

Another possible approach to address resilience is through fault-tolerant algorithms. Addressing resilience through AMR replication would be a cost-effective approach as it requires only a few modifications to the existing algorithms.

As discussed in the Inter-Agency Workshop on HPC Resilience at Extreme Scale [3], existing approaches use checkpointing to address fault-tolerance, and further research has been focused on achieving faster checkpointing. System software must be given an opportunity to recover from an error and to carry on with the computation. A similar approach is followed in our system. When a node dies, thereby causing data loss, we recover the lost data using interpolation routines and the computation is carried on.

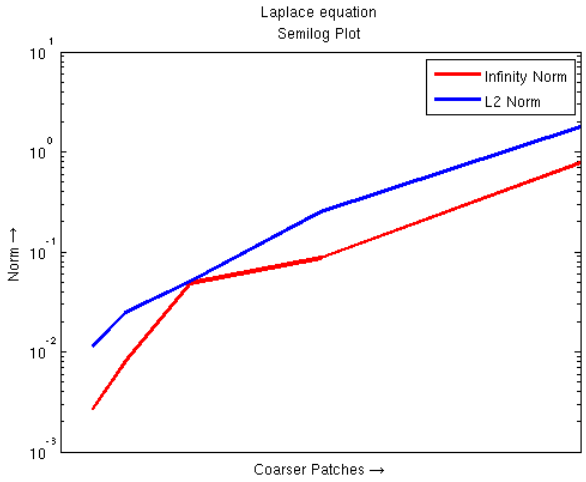

Figure 11: Norm vs Patch Resolution

Several experiments were conducted where different levels of coarser and finer meshes were used in interpolation routines and the accuracy of the interpolated data was observed. For the study, we chose the solution of Laplace Equation and Shepard Interpolation to preserve the positivity of the data [16]. We noticed that as the coarseness of the patch increases, the error increases which is expected. It is interesting to note the amount of increase in error as the coarseness of the patch increases. In this approach, a coarser version of the patch is replicated. Thus in 3D, this approach has only 12.5% overhead as suggested by Meng [39], Heroux [27] and others.

N x N x N grid cells were taken and assuming that the center patch of p x p x p grid cells is missing, the lost data is recovered using the interpolation routine from the remaining data. This experiment was repeated with coarser meshes of size $[\frac{N}{2}]^3$, $[\frac{N}{4}]^3$,$[\frac{N}{8}]^3$, and the center patch of p x p x p grid cells was recovered through interpolation. Figure 11 shows the variation of the norm with increasing coarseness of the patch. The infinity norm and L^2 Norm are shown along the y-axis. Patch coarseness is represented along the x-axis.

The obvious advantage of using AMR replication is that it adds an overhead of only 12.5% to the amount of storage when the patches are one level coarser. In the case of time integration, it may be possible to integrate coarse patches, twice as large as the fine mesh, within a time step, which reduces the cost of further time integration. The disadvantage is that every time a patch fails there is the equivalent of a spatial re-mesh. The challenge is then ensuring that the fine patch is recreated with enough accuracy from the coarse patch. The key to this process is performing spatial interpolation with enough accuracy. An analysis of the errors introduced in this process is given in [12]. The analysis shows that interpolation accuracy is critical after re-meshing restarts. In our case, this applies also to the interpolation after a patch crashes.

6.2 Power Management Systems

When considering the proposed 20 MW power cap adopted by the Exascale Initiative Steering Committee to reach exascale [9], it is important that a resilience strategy not restrict itself to hardware and software failures alone. Due to available power budgets and restrictions, it may be necessary for computation to be "throttled down" at times. Power management approaches such as the PowerAPI [19] from Sandia National Laboratories, seek to provide a portable API for power measurement and control to accomplish

this. It is commonly considered that, through this measurement and control, exascale energy efficiency requirements will be met. As proposed in [19], dynamic runtime adaptation is one anticipated system-level approach to power management, making throttling decisions based on power and performance counter data. In this approach, clock modulation is used to reduce power usage on one or more cores per chip. The Uintah runtime system is well positioned to adapt to an approach such as this because of its task-based design.

In this work, we have added the Uintah facility to artificially slowdown hardware, emulating these power reduction scenarios. The effect of this facility ranges from individual cores to entire nodes within a running simulation, and allows programmatic randomization of a subset of the overall ranks involved in `MPI_COMM_WORLD`. For these ranks, `sleep()` calls are injected in between the end of task execution and the task finalization stage in such a way that this delay is seen by the Uintah load balancer. The key aim is to see what the effect of either a core or complete nodal slowdown has on the Uintah load balancer, specifically how Uintah's load balancing algorithms [32] respond to slower execution by certain MPI ranks.

Table 3: Effect of randomized slowdown of randomized MPI ranks

GT	.01	.025	.50	.1	.15	.2	.25	.3
Time (s)	594	397	292	406	446	424	819	1178

Table 3 shows preliminary results for a single-material, single-level ICE problem with initially uniform workload across computational domain with 8 MPI ranks and a computational domain decomposed into 25 mesh patches. 2 slow ranks were selected at the beginning of the simulation, which ran for 100 timesteps. The Kalman cost algorithm was used and `GainThreshold` (GT) of the load balancer varied from 0.01 to 0.3. Load balancing was triggered every 5 timesteps. The table illustrates the ability of the nodal slowdown facility to impact the Uintah load balancer. We are currently exploring how and when patches are migrated off "slow nodes".

7. RELATED WORK

There has been much recent activity in the area of task-based approaches that also use run time systems to address issues such as portability and resilience. One of the the most widely used task based approaches is the Charm++ system of Kale and co-workers which has used a checkpointing approach since 2004 [51]. There are similar initiatives in DOE labs and other often connected efforts to build parallel system software for exascale machines. The design of modern operating systems such as ARGO [1] and Hobbes [15], custom languages such as Habanero language [18, 50] and The Sandia Xpress Xstack Project involving Sterling's ParallelX approach [31] are very promising ideas for exascale computing. In addition, there are many papers about the challenges at exascale such as [17, 48] which give a detailed discussion on the current state and approaches to building resilience.

The work of the EXAHD project uses a two-level sparse grid approach for higher-dimensional problems [44, 46]. This study proved that we might not need the entire data for recovery procedures. Extending the same idea, we use a coarser grid for checkpointing as we do not need the complete data for recovery.

The Inria RUNTIME team led by Namyst with his colleagues are developing a suite of tools and software layers including StarPU [4–8,25,26,35,36,49] for task based runtime systems that work across the heterogeneous platforms of exoscale HPC systems.

8. CONCLUSIONS AND FUTURE WORK

Future Uintah run time system development will entail continued development and implementation of the described design for supporting multiple GPUs per node and assigning a single MPI rank per node to minimize latencies and memory management overhead. We will continue to leverage the capabilities of the Kokkos layer to handle the changes and challenges of multiple memory hierarchies and various accelerator architectures in an agnostic way. Full support of Kokkos will require modifying Uintah's data warehouse to return Kokkos views. Data structure improvements will focus on eliminating the storage of the entire patch decomposition per MPI rank. The cost of storing and processing all of these patches per MPI rank limits the scalability at exascale problem sizes. The ultimate solution is to store a subset of the patches per MPI rank and communicate information as needed. While simulating core slowdown through software is an effective means for exploring approaches for handling resiliency and redundancy in the runtime system, we will augment simulated hardware failures at the core/node levels with actual deployment and testing on hardware that can be controlled programatically such as found on the Sandia test bed machines. In addition, we will continue to investigate the use of RedMPI as a toolkit for exploring the injection of faults into the runtime system layer to stress test our approaches to redundancy and resiliency management.

The simulation to test core failure has two subtle assumptions: cores don't fail during a reduction task and the main core doesn't fail. Both these constraints will be removed to handle core failures at any stage of simulation. While the cores have been programmed to fail randomly, we plan to improve the prediction capabilities of the runtime system to trigger the code to handle core failure.

Finally, the resilient system that we are trying to build must dynamically adapt to the system errors with minimal hardware support. Better algorithms for interpolation including radial basis functions, and Kriging will be investigated as a means to recover from errors such as data loss that obviate a checkpoint restart. The recovery algorithms should be faster than restarting from a checkpoint and so help greatly in the case of predicted frequent faults.

9. ACKNOWLEDGMENTS

This work has been supported by the Department of Energy, National Nuclear Security Administration, under Award Number DE-NA0002375 and by the National Science Foundation XPS award number 1337145.

10. REFERENCES

[1] Argo: An exascale operating system. http://www.mcs.anl.gov/project/argo-exascale-operating-system.

[2] Portable hardware locality. http://http://www.open-mpi.org/projects/hwloc/.

[3] B. Adolf, S. Borkar, N. DeBardeleben, M. Elnozahy, M. Heroux, D. Rogers, R. Ross, V. Sarkar, M. Schulz, M. Snir, and P. Woodward. Inter-agency workshop on hpc resilience at extreme scale. National Security Agency, Advanced Computing Systems, February 2012.

[4] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In Wen-mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, Sept. 2010.

[5] E. Agullo, O. Beaumont, L. Eyraud-Dubois, J. Herrmann, S. Kumar, L. Marchal, and S. Thibault. Bridging the Gap between Performance and Bounds of Cholesky Factorization on Heterogeneous Platforms. In *Heterogeneity in Computing Workshop 2015*, Hyderabad, India, May 2015.

[6] C. Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, Dec. 2011.

[7] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, Feb. 2011.

[8] S. Benkner, S. Pllana, J. L. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov. PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems. *IEEE Micro*, 31(5):28–41, Sept. 2011.

[9] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick. Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor and study lead, 2008.

[10] M. Berzins. Status of release of the Uintah Computational Framework. Technical Report UUSCI-2012-001, Scientific Computing and Imaging Institute, 2012.

[11] M. Berzins, J. Beckvermit, T. Harman, A. Bezdjian, A. Humphrey, Q. Meng, J. Schmidt, and C. Wight. Extending the uintah framework through the petascale modeling of detonation in arrays of high explosive devices. *Submitted to SIAM Journal on Scientific Computing*, 2015.

[12] M. Berzins, P. Capon, and P. Jimack. On spatial adaptivity and interpolation when using the method of lines. *Applied Numerical Mathematics*, 26:117–134, 1998.

[13] M. Berzins, J. Luitjens, Q. Meng, T. Harman, C. Wight, and J. Peterson. Uintah - a scalable framework for hazard analysis. In *TG '10: Proc. of 2010 TeraGrid Conference*, New York, NY, USA, 2010. ACM.

[14] M. Berzins, J. Schmidt, Q. Meng, and A. Humphrey. Past, present, and future scalability of the uintah software. In *Proceedings of the Blue Waters Extreme Scaling Workshop 2012*, page Article No.: 6, 2013.

[15] R. Brightwell, R. Oldfield, A. B. Maccabe, and D. E. Bernholdt. Hobbes: Composition and virtualization as the foundations of an extreme-scale os/r. In *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '13, pages 2:1–2:8, New York, NY, USA, 2013. ACM.

[16] K. W. Brodlie, M. R. Asim, and K. Unsworth. Constrained visualization using the shepard interpolation family. Technical report, SCHOOL OF COMPUTING, UNIVERSITY OF LEEDS, LEEDS, 2005.

[17] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. *Supercomputing frontiers and innovations*, 1(1), 2014.

[18] V. Cavé, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: The new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, PPPJ '11, pages 51–61, New York, NY, USA, 2011. ACM.

[19] D. DeBonis, R. E. Grant, S. L. Olivier, M. Levenhagen, S. M. Kelly, K. T. Pedretti, and J. H. Laros. A power api for the hpc community. Sandia Report SAND2014-17061, Sandia National Laboratories, 2014.

[20] D.L.Brown and P. et al. Scientific grand challenges: Crosscutting technologies for computing at the exascale. Technical Report Report PNNL 20168, US Dept. of Energy Report from the Workshop on February 2-4, 2010 Washington, DC, 2011.

[21] C. Earl. *Introspective Pushdown Analysis and Nebo*. School of Computing, University of Utah, 2014.

[22] H. C. Edwards, C. R. Trott, and J. Amelang. Kokkos tutorials, 2015. https://github.com/kokkos/kokkos-tutorials.

[23] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[24] D. Fiala. Detection and correction of silent data corruption for large-scale high-performance computing. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 2069–2072, May 2011.

[25] S. Henry. *Modèles de programmation et supports exécutifs pour architectures hétérogènes*. PhD thesis, Université Bordeaux 1, 351 cours de la Libération — 33405 TALENCE cedex, Nov. 2013.

[26] S. Henry, A. Denis, and D. Barthou. Programmation unifiée multi-accélérateur OpenCL. *Techniques et Sciences Informatiques*, (8-9-10):1233–1249, 2012.

[27] M. Heroux. Untitled manuscript. personal communication.

[28] A. Humphrey, T. Harman, M. Berzins, and P. Smith. A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing. In J. M. Kunkel and T. Ludwig, editors, *High Performance Computing*, volume 9137 of *Lecture Notes in Computer Science*, pages 212–230. Springer International Publishing, 2015.

[29] A. Humphrey, Q. Meng, M. Berzins, and T. Harman. Radiation modeling using the Uintah heterogeneous CPU/GPU runtime system. In *Proceedings of the 1st Conference of the Extreme Science and Engineering Discovery Environment (XSEDE 2012)*. ACM, 2012.

[30] J.Ang and K. et al. Workshop on extreme-scale solvers: Transition to future architectures. Technical Report USDept. of Energy, Office of Advanced Scientific Computing Research. Report of a meeting held on March 8-9 2012, Washington DC, 2012.

[31] H. Kaiser, M. Brodowicz, and T. Sterling. Parallex an advanced parallel execution model for scaling-impaired applications. In *Parallel Processing Workshops, 2009. ICPPW '09. International Conference on*, pages 394–401, Sept 2009.

[32] J. Luitjens. *The Scalability of Parallel Adaptive Mesh Refinement Within Uintah*. PhD thesis, School of Computing, University of Utah, 2011. Advisor: Martin Berzins.

[33] J. Luitjens and M. Berzins. Improving the performance of Uintah: A large-scale adaptive meshing computational framework. In *Proc. of the 24th IEEE Int. Parallel and Distributed Processing Symposium (IPDPS10)*, 2010.

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

[35] S. A. Mahmoudi, P. Manneback, C. Augonnet, and S. Thibault. Traitements d'images sur architectures parallèles et hétérogènes. *Technique et Science Informatiques*, 2012.

[36] V. Martínez, D. Michéa, F. Dupros, O. Aumage, S. Thibault, H. Aochi, and P. O. A. Navaux. Towards seismic wave modeling on heterogeneous many-core architectures using task-based runtime system. In *27th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, Florianopolis, Brazil, Oct. 2015.

[37] C. D. Martino, Z. Kalbarczyk, R. K. Iyer, F. Baccanico, J. Fullop, and W. Kramer. Lessons learned from the analysis of system failures at petascale: The case of blue waters. In *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '14, pages 610–621, Washington, DC, USA, 2014. IEEE Computer Society.

[38] M. L. Massie, B. N. Chun, and D. E. Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817 – 840, 2004.

[39] Q. Meng. Untitled manuscript. personal communication.

[40] Q. Meng. *Large-scale distributed runtime system for DAG-based computational framework*. PhD thesis, University of Utah, Salt Lake, 2014.

[41] Q. Meng and M. Berzins. Scalable large-scale fluid-structure interaction solvers in the Uintah framework via hybrid task-based parallelism algorithms. *Concurrency and Computation: Practice and Experience*, 2013.

[42] Q. Meng, M. Berzins, and J. Schmidt. Using Hybrid Parallelism to Improve Memory Use in the Uintah Framework. In *Proc. of the 2011 TeraGrid Conference (TG11)*, Salt Lake City, Utah, 2011.

[43] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating applications portability with the Uintah DAG-Based runtime system on PetScale supercomputers. In *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 96:1–96:12. ACM, 2013.

[44] A. Parra Hinojosa, C. Kowitz, M. Heene, D. Pflüger, and H.-J. Bungartz. Towards a fault-tolerant, scalable implementation of gene. In *Proceedings of ICCE 2014*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2015. Accepted.

[45] B. Peterson, H. Dasari, A. Humphrey, J. Sutherland, T. Saad, and M. Berzins. Reducing overhead in the uintah framework to support short-lived tasks on gpu-heterogeneous architectures. In *Under submission to WOLFHPC Workshop - International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '15, 2015.

[46] D. Pflüger, H.-J. Bungartz, M. Griebel, F. Jenko, T. Dannert, M. Heene, A. Parra Hinojosa, C. Kowitz, and P. Zaspel. Exahd: An exa-scalable two-level sparse grid approach for higher-dimensional problems in plasma physics and beyond. In *Euro-Par 2014: Parallel Processing Workshops*, Lecture Notes in Computer Science. Springer-Verlag, Sept. 2014. accepted.

[47] R. Rajachandrasekar, X. Besseron, and D. K. Panda. Monitoring and predicting hardware failures in hpc clusters with ftb-ipmi. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1136–1143, Washington, DC, USA, 2012. IEEE Computer Society.

[48] M. Snir, R. Wisniewski, J. Abraham, S. Adve, S. Bagchi, P. Balaji, J. Belak, P. Bose, F. Cappello, B. Carlson, A. Chien, P. Coteus, N. Debardeleben, P. Diniz, C. Engelmann, M. Erez, S. Fazzari, A. Geist, R. Gupta, F. Johnson, S. Krishnamoorthy, S. Leyffer, D. Liberty, S. Mitra, T. Munson, R. Schreiber, J. Stearley, and E. Hensbergen. Addressing Failures in Exascale Computing. *International Journal of High Performance Computing Applications*, March 21, 2014 2014.

[49] L. Stanisic, S. Thibault, A. Legrand, B. Videau, and J.-F. Méhaut. Faithful Performance Prediction of a Dynamic Task-Based Runtime System for Heterogeneous Multi-Core Architectures. *Concurrency and Computation: Practice and Experience*, page 16, May 2015.

[50] S. Tasirlar and V. Sarkar. Data-driven tasks and their implementation. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 652–661, Washington, DC, USA, 2011. IEEE Computer Society.

[51] G. Zheng, L. Shi, and L. Kale. Ftc-charm++: an in-memory checkpoint-based fault tolerant runtime for charm++ and mpi. In *Cluster Computing, 2004 IEEE International Conference on*, pages 93–103, Sept 2004.