An Evaluation of An Asynchronous Task Based Dataflow Approach For Uintah

1st Alan P. Humphrey
SCI Institute
University of Utah
Salt Lake City, Utah
ahumphrey@sci.utah.edu

2nd Martin Berzins
SCI Institute
University of Utah
Salt Lake City, Utah
mb@sci.utah.edu

***Abstract*—The challenge of running complex physics code on the largest computers available has led to dataflow paradigms being explored. While such approaches are often applied at smaller scales, the challenge of extreme-scale data flow computing remains. The Uintah dataflow framework has consistently used dataflow computing at the largest scales on complex physics applications. At present Uintah contains two main dataflow models. Both are based upon asynchronous communication. One uses a static graph-based approach with asynchronous communication and the other uses a more dynamic approach that was introduced almost a decade ago. Subsequent changes within the Uintah runtime system combined with many more large scale experiments, has necessitated a reevaluation of these two approaches, comparing them in the context of large scale problems. While the static approach has worked well for some large-scale simulations, the dynamic approach is seen to offer performance improvements over the static case for a challenging fluid-structure interaction problem at large scale that involves fluid flow and a moving solid represented using particle method on an adaptive mesh.**

***Index Terms*—DataFlow, Asynchronous, Uintah, Runtime, Scalability**

I. Introduction

In the realm of scientific and engineering computing an important class of dataflow algorithms and software are those based upon the Asynchronous Many Task (AMT) paradigm. Such architectures date back at least to [27] and have the flexibility to handle both heterogeneous hardware architectures and complex applications at large scale. An AMT program consists of a flow of data control and data relationships between tasks within the AMT model. Control of the dataflow is through a runtime system that uses a dynamic directed acylic graph (DAG) to guide task execution. Unlike bulk synchronous parallel (BSP) approaches with a fixed execution order, the AMT runtime extracts the appropriate level of parallelism by automatically mapping tasks to available computational resources. Specific examples are, Charm++ [14], Legion [32], DHARMA [1], Uintah [19], STAPL [33], OCR [34], Parsec [6], StarPU [35], HTGS [36], and HPX [37]. Sterling summarizes these and other examples in [26]. In this work the effectiveness of this AMT approach is considered in the context of the Utah open source Uintah software. The heart of Uintah is a sophisticated computational framework that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and execute the resulting multi-physics simulation [5], [19]. The particular application focus of Uintah has made possible an early transition to large scale applications [3], [10], [15], [28]. This development of Uintah has been continuous since

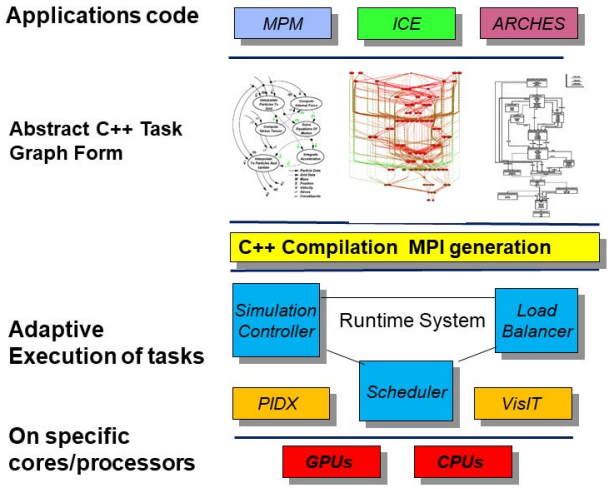

Fig. 1. Outline of Uintah Architecture.

1998 through the University of Utah DOE Center for the Simulation of Accidental Fires and Explosions (C-SAFE) [5] (9/97 – 3/08), and then more recently from 2014 to 2020 through the DOE NNSA funded CCMSC Center in Utah. In this latter case Uintah is being used to model a very large clean coal boiler with turbulent combustion [15], [29]. Uintah solves complex systems of partial differential equations (PDEs) on Structured AMR (SAMR) grids with a number of applications codes and a runtime system that can integrate multiple simulation components, analyze the dependencies and communication patterns between them, and efficiently execute the resulting multi-physics simulation. The architecture of Uintah is shown in Figure I. Uintah presently contains five main simulation algorithms, or components, and many user specific packages: 1) the ICE compressible multi-material finite-volume CFD component, 2) the particle-based Material Point

978-1-7281-2607-4/19/$31.00 ©2019 IEEE
DOI 10.1109/COMPSAC.2019.10282

Method (MPM) [4] for structural mechanics, 3) the combined fluid-structure interaction (FSI) algorithm MPM-ICE [10], 4) the Arches turbulent reacting CFD component [28], which uses hypre [30] to solve the pressure Poisson equation that arises from the low mach number approximation [25] and 5) the multi-physics, platform-portable reacting flow component Wasatch [23]. Three of these components are shown in Figure I. In addition there are many informal software applications components written by specific application groups. Each of these components generates a set of interconnected tasks. These tasks are then "compiled" by Uintah into a task graph where the MPI messages to connect the tasks are automatically generated. The task graph is then executed by a scheduler, either in a static or a dynamic fashion as is described below. For visualization Uintah relies on the Visit package while for fast I/O, PIDX [15] is used. Finally, while a discussion of this is out of the scope of this work there is a major effort underway to introduce a performance portability layer into Uintah through the use of Kokkos [21].

II. The Original Uintah Dataflow Framework

Uintah's dataflow model utilizes an abstract task graph representation of parallel computation and communication to express data dependencies between multiple physics components and to schedule work. This dataflow approach is similar to Charm++ [14], but has its own different and distinctive approach. For example, Uintah uses a "Data Warehouse", a distributed data store for simulation variables through which all MPI data transfer takes place. Uintah's Data Warehouse abstraction ensures that the user-coded tasks are independent of the hidden communications MPI layer. Each Uintah component specifies a list of tasks to be performed and the data dependencies between them in a declarative style. Each task has a C++ method which is used to perform the actual computation, and consumes some input and produces some output (which is in turn the input of some future task). The task graph is a directed acyclic graph of tasks that allows Uintah to analyze the structure of the computation to automatically enable load-balancing, data communication, parallel I/O, and checkpoint/restart. The Uintah task scheduler compiles all tasks and variable dependencies into a task graph and is responsible for, 1) computing the dependencies of tasks, 2) determining the order of execution, and 3.) ensuring that the correct inter-process communication is performed. Dependency edges are added between tasks based on the supplied variable dependencies. The computed dependency edges can be either internal or external. Internal dependencies are between patches on the same processor and external dependencies are between patches on different processors. *Thus internal dependencies imply a necessary order where external dependencies specify required communication.* The compilation process also combines external dependencies from the same source or to the same destination. The Uintah scheduler automatically sets up MPI communication for data dependencies between MPI processes. When a task completes, its outputs are sent to other tasks that require them. A load balancer component is responsible for assigning each patch to a processor. Uintah's load balancer utilizes space-filling curves in order to cluster spatially contiguous patches together [16] with a predictive workload approach. Initially, a static task execution approach was employed and was successful for the C-SAFE [5] research of Uintah's first decade.

III. Uintah's Asynchronous Runtime System

Moving Uintah to larger machines and to applications that have dynamically varying task graphs, such as adaptive mesh refinement (AMR), exposed limitations in the static execution paradigm. In particular, achieving scalability for AMR based simulations, such as changing the grid in response to a solution evolving in time, requires regridding, load balancing and task scheduling [16] whenever regridding occurs. Poor performance in any of these steps can lead to performance problems at larger scales [8], [9], [16].

Results in [17] showed that with static execution there was a substantial increase in MPI communication time at larger numbers of cores. This was particularly the case on fluid-structure simulations for which some mesh patches had much more work due to the presence of the solid material as modeled by MPM particles in parts of the domain and not in others. Measurements showed that the delay in a task waiting for an MPI message was nearly 80% of the total MPI waiting time in Uintah. The new dynamic scheduler introduced by [17] changes the task order during the execution to overlap communication and computation. While substantial development was required to support the dynamic and out-of-order execution, there was a significant performance benefit in lower MPI wait time and overall scalability. The runtimes were higher however [17]. The dynamic scheduler utilizes two task queues: an internal ready queue and an external ready queue. If a task's internal dependencies are satisfied, then that task will be put in the internal ready queue where they will wait until all required MPI communication has finished. A counter of outstanding MPI messages is tracked for each task. When this counter reaches zero, the communication is complete and the task is ready to be executed. At that point it is placed in the external ready queue. When scheduling a task the scheduler chooses a task in the external ready queue based on a priority algorithm. This process is illustrated in Figure 2. As long as the external queue is not empty, the processor always has tasks to run. This can help to overlap the MPI communication time with task execution. This approach reduces MPI wait times significantly [31].

There is one task graph per mesh patch and these graphs are coupled either by MPI communications or internally to a node. If there are more patches than cores, this means that tasks associated with a patch in the interior of a domain may start executing while tasks with external communications have to wait for MPI transmitted halo elements for example, as shown in Figure 3 in which the central four patches are assumed to have halo values that reside on the same node. In effect an over-decomposition approach is used because of the distributed and local nature of the patch task graphs. In

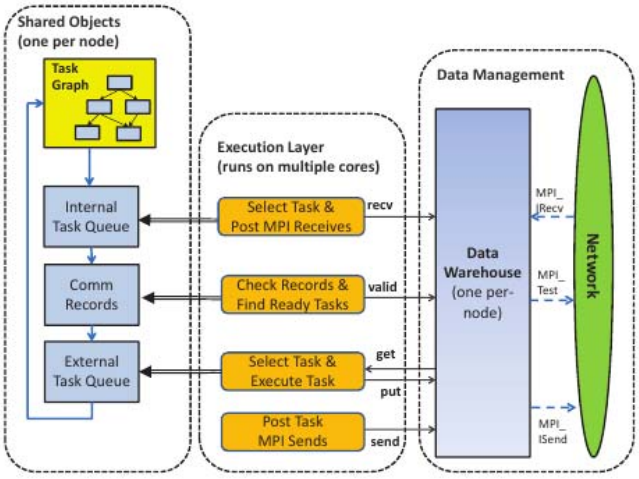

Fig. 2. Uintah Nodal Architecture

particular, in the dynamic approach that the tasks on a node may execute on any of the cores on that node. In contrast with the static approach of the MPI Scheduler one fixed mesh patch is assigned to a core and thus the exterior cores have to wait for communications to arrive from other nodes.

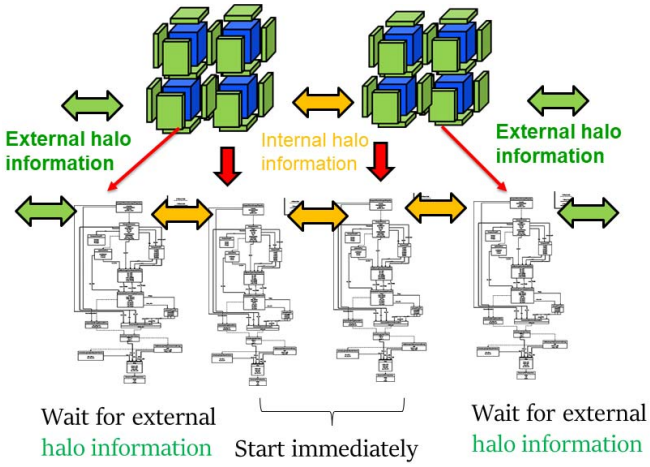

Fig. 3. Uintah's Patch-Based DAG structure.

IV. Uintah's Present Status and Runtime System

As a result of continuous development, Uintah has a number of distinctive features:

- A shared memory [17] approach that is lock-free, is implemented by making use of atomic operations and thus allows efficient access by all cores to the shared data on a node. This involved a re-design of the shared data structures so as to make them lock-free by using hardware-based atomic operations, (as in modern CPUs);
- Decentralized execution [17] of the task graph is implemented by each CPU core requesting work itself as this eliminates having a single controlling thread on a core and under use or contention for it;
- Accelerator task execution [20], [22] on a node is implemented through an extension of the runtime system that enables tasks to be executed efficiently (through preloading of data) on one or more accelerators per node and through rethinking data structures in the data warehouse;
- A long-established and lightweight trace system that allows identification of the scalability properties of different parts of the code;
- The mesh refinement strategy of Uintah is used not only for standard adaptive meshing [16] but also for a novel approach to scalable ray-tracing radiation calculations [12], as well as to provide a resilience strategy, in that coarse versions of patch data are stored on other nodes to allow for recovery without checkpointing should a node crash [24];
- The scalability of Uintah has been achieved where needed through exploiting asynchronous (including out-of-order) task execution, over-decomposition of tasks, overlapping of communication and computation, work stealing task graph prioritization based upon communication needs and dependencies.

As well as the conventional CPU results shown, Uintah has been used on GPU and Xeon Phi architectures [11], [12], [20], and runs on both NSF, DOE, and other parallel computers (Stampede, Stampede2, Mira, Titan, and Sunway TaihuLight), and on many smaller clusters worldwide. With access to larger and more diverse computational environments, the scalability of the Uintah framework has been extended, necessitating continual improvements in the framework itself.

There are two schedulers at present within Uintah that use the approaches described in the previous section. The first is known as the MPI Scheduler. This scheduler uses a *pure MPI* or *"MPI-only"* approach, e.g., one MPI process per CPU core, and employs a fixed task execution pattern for the task graph. *This may be thought of as a static dataflow approach*, as the execution pattern is derived from a topological sort of the underlying task graph, with "fixed" connections between tasks. Execution order of computational tasks is deterministic. It is however important to note that because of the still-asynchronous nature of nonblocking MPI communications under this model, this approach is *NOT* a bulk synchronous approach. The use of such an approach would be even less efficient that the static task execution with asynchronous communication approach used here. Nevertheless, the use of a fixed execution pattern means that there may be delays in tasks executing due to MPI wait times as was observed by Meng, et al. [17]. Another limitation of this pure MPI scheduling is the that task data dependencies have to be passed through MPI messages and copied to another process' memory even if the source and destination tasks were on the same multi-core node.

The need to improve Uintah's scalability has led to the adoption of a nodal shared memory model in which there is only one MPI process per multicore node (or NUMA node),

one global memory per node and task execution on individual CPU cores is through Pthreads (`std::thread`) bound to available cores on-node (MPI + X). This approach has led to a second, *multithreaded* scheduler, known as the Unified Scheduler. This scheduler has also helped made it possible to reduce nodal memory footprint by a factor of 10 due to the elimination of redundant halo information and global metadata. Additionally, this approach eliminates the intranodal MPI imposed by the pure MPI Scheduler. With the Unified Scheduler, it is possible to achieve dynamic, out-of-order execution of tasks (out of order with respect to the topological sort used by the MPI scheduler). *This model may be viewed as a dynamic dataflow approach* in that task execution is *nondeterministic*. The connectivity edges within the task graph are *not* static, and task execution order is determined solely by message arrival times. This variability in task execution order is evident on differing communications networks. The present Unified Scheduler builds upon the original version of [17] with the internal and external ready queues, for example, but has been substantially revised in the course of recent work. The Unified Scheduler is also the only task scheduler that supports GPU tasks. There are many important and substantial differences however, primarily that 1) significant infrastructure changes have been required for thread-safe access to shared data structures, namely ensuring that asynchronous access to the data warehouse is not potentially catastrophic, and 2) task execution is based on external data flow (MPI), specifically message arrival times, in other words when external dependencies are met for a particular task. Achieving correct, thread-safe access to Uintah's Data Warehouse has required the carefully combined use of atomic operations and other C++11 synchronization primitives. This has not only been required in the Data Warehouse, but also for shared access to Uintah's task queues and within its MPI engine for things such as the external dependency (MPI message) count.

Part of the challenge with a nodal shared memory model is the selection, design, and usage of data structures, language constructs, and synchronization primitives that 1.) achieve correctness and thread safety, and 2.) do not create performance bottlenecks due to unnecessary locks, coarse-grained critical sections, and other serialization points within the code. For Uintah, which currently uses `MPI_THREAD_MULTIPLE` when employing the Unified Scheduler, allows individual threads to perform their own MPI sends, receives, and collective operations. This approach, in addition to shared access to nodal task queues and Data Warehouse, uses mixed concurrency models (MPI + std::thread + CUDA), a situation that has the potential for problematic race conditions and deadlock scenarios, some of which only manifest at larger scale in our experience. Overly synchronized code can significantly degrade or even negate the performance benefits gained from a dataflow approach with asynchronous communication.

Prior to the results shown in Table II when running this challenging FSI problem at large scale, significant performance degradation was observed within the Unified Scheduler. This slowdown was 2.5-3X relative to the MPI Scheduler running the same problem at scale, and warranted exploration. Several synchronization and thread safety issues were discovered within the Uintah infrastructure and addressed in this work:

- Unnecessary locks and other synchronization constructs within Uintah's DataWarehouse and MPI engine that were replaced with simple usage of `std::atomic<T>` and other built-in compare-and-swap (CAS) instructions;
- Inadvertent nested task queue locks;
- Unnecessary critical sections, that were still protected by a mutex (`std::mutex`), likely remnants of earlier design ideas not completed;
- Overly coarse-grained critical sections that were either unnecessary or able to be made more fine-grained, some of which were in the code critical path, e.g., processing of MPI receives, which for `MPI_THREAD_MULTIPLE`, encounters locks within the MPI library – a significant serialization point for code that sees heavy thread traffic.

The resolution of these issues enabled the results in Table II to be obtained. Due to the clear separation between Uintah applications and its runtime, applications automatically benefit from these runtime improvements.

V. Evaluating DataFlow Approaches

In evaluating the static versus dynamic approaches on more modern architectures than those used by Meng [17] requires the status of the MPI and Unified schedulers to be explained here. The principal contribution of this work is in the improvements to the Unified Scheduler in order to overcome the 2.5-3X slowdown over the MPI scheduler shown by Meng in [17] on this FSI benchmark problem.

A. MPI Scheduler

The MPI Scheduler has been used on the DOE Mira architecture to achieve weak and strong scaling up to 512K CPU cores [15] on the massive turbulent combustion coal boiler simulations problems solved by the Utah CCMSC Center. The hypre linear solver package [30] employed to solve the linear systems that arise in this problem, as expected, only weak scales and does not strong scale [15]. As this solver is also used in some radiation calculations this is potentially problematic, from a strong scaling point of view. While the MPI scheduler uses fixed execution it seems to not be sensitive to the task execution paradigm and depends more on the asynchronous MPI approach used by Uintah for scalability and performance. It is only at extreme scales and with very dynamic workloads that we see a breakdown in scalability with the fixed execution approach used by the MPI scheduler. In terms of the experiments using the FSI problem in Section VI, the principal breakdown in scalability for the MPI scheduler is due to the fact that the computing cost of a patch with particles is roughly six times that of computing a patch without particles. This introduces a significant load imbalance that leads to poor scaling due to the fact that load balancing a region with particles, even with measurement-based load balancing, is difficult to predict. In the case of the MPI

scheduler, if one CPU core (a single MPI process) has finished its assigned work faster than other cores, it idly waits for other cores to finish, even if these cores are on the same node [17].

B. Unified Scheduler

The Unified Scheduler has been employed on the same CCMSC target boiler problem, when run with a reverse Monte Carlo ray tracing (RMCRT) radiation solver [12]. The excellent scaling seen with the radiation component on its own translates to the whole boiler. Table I shows the mean time per timestep (averaged over 10 timesteps involving a radiation solve) from 16,384 to 262,144 CPU cores (1k to 16K single, Nvidia K20 GPUs, respectively). The super linear scaling is achieved as a result of the GPU kernels being executed at maximum efficiency. As mentioned in Section IV, the Unified Scheduler is the only Uintah task scheduler with GPU task support, hence results for the MPI Scheduler are not shown. These results show strong scaling out to near the full extent of DOE Titan for a full production boiler case with Arches and GPU-based RMCRT radiation solve, and they are the first such results of such a complex geometry coupled to a full combustion model [13].

TABLE I
UNIFIED SCHEDULER: COAL BOILER ON DOE TITAN

Cores	16K	32K	64K	128K	256K
GPUs	1K	2K	4K	8K	16K
Time	821	407	203	99	55

These results show excellent strong scaling for the whole code. This is in part due to the time spent by the hypre linear solver being a small part of the overall cost. Further details are in Humphrey [13]. Humphrey also shows that with careful use of mesh refinement it is possible to get the adaptive mesh ray tracing code to weak scale [13]. It is thus one of the few radiation algorithms with good strong and weak scaling properties for these thermal radiation problems in which every mesh cell is a potential radiation source.

VI. COMPARING STATIC AND DYNAMIC EXECUTION ON AN FSI BENCHMARK

The benchmark problem used to compare the two schedulers is the Fluid-Structure Interaction Benchmark developed by Meng [17], [18]. This problem simulates a moving solid through a domain filled with air to represent key features of our benchmark problems as modeled using a combination of a fluid solver and a particle method with adaptive meshing (the MPMICE algorithm) in the Uintah framework. The differences between the two schedulers are shown using the two separate resolutions in [3] for our benchmark problem, *resolution-A* (192^3 cells) and *resolution-B* (384^3 cells).

For the *resolution-A* case, three refinement levels are used for the simulation grid with each level being a factor of four more refined than the previous level. This problem has a total of 3.62 billion particles, 518 million cells and 277,778 total patches created on three AMR levels. While our benchmark problem with *resolution-A* achieved excellent scalability to 512K CPU cores on the DOE Mira system [18], we observed a significant breakdown in scaling at 768K CPU cores due to there being less than 0.3 patches per core and hence devised a much larger resolution problem.

This FSI problem, *resolution-B*, has a resolution of (384^3 cells) by doubling the resolution in each direction resulting in nearly an order of magnitude increase in problem size. The *resolution-B* problem uses a grid with three mesh refinement levels, with each level being a factor of four more refined than the previous level, and has a total of 29.45 billion particles, 3.98 billion cells created on three AMR levels, and 1.18 million total patches.

The results shown in Table II show that for both these resolutions, the new Unified Scheduler is both faster and appears to scale better. For example with *resolution-B* at the largest core count the static approach is 50% slower than the asynchronous method. Overall the Unified Schedular now has the potential to be the default scheduler for Uintah.

TABLE II
FSI PROBLEM UNIFIED VS MPI SCHEDULER ON DOE TITAN

	Res A		Res B	
Cores	MPI	UNI	MPI	UNI
32K	17	13.5	-	-
64K	12.8	7.5	77.5	65.1
128K	10.3	5.0	36.7	31.0
256K	8.2	3.6	26.2	17.6

VII. CONCLUSIONS

We have looked at the performance of two different dataflow approaches to task execution within the Uintah software. While the combination of static task execution and asynchronous MPI messaging has proven to be reasonably effective in some situations, for complex problems such as the fluid-structure interaction problem shown here, the use of dynamic asynchronous task execution offers clear performance benefits.

VIII. ACKNOWLEDGEMENTS

This material is based upon work supported by the Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002375. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. An award of computer time was provided by the DOE ALCC Program. The authors would like to thank all those involved with Uintah past and present.

REFERENCES

[1] J. Bennett, R. Clay, G. Baker, M. Gamell, D. Hollman, S. Knight, H. Kolla, G. Sjaardema, N. Slattengren, K. Teranishi, J. Wilke, M. Bettencourt, S. Bova, K. Franko, P. Lin, R. Grant, S. Hammond, S. Olivier, L. Kale, N. Jain, E. Mikida, A. Aiken, M. Bauer, W. Lee, E. Slaughter, S. Treichler, M. Berzins, T. Harman, A. Humphrey, J. Schmidt, D. Sunderland, P. McCormick, S. Gutierrez, M. Schulz, A. Bhatele, D. Boehme, P. Bremer, T. Gamblin. ASC ATDM level 2 milestone 5325: Asynchronous many-task runtime system analysis and assessment for next generation platforms, Sandia National Laboratories, 2015.

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

[3] Berzins, M., Beckvermit, J., Harman, T., Bezdjian, A., Humphrey, A., Meng, Q., Schmidt, J., Wight, C. Extending the Uintah Framework through the Petascale Modeling of Detonation in Arrays of High Explosive Devices, SIAM Journal on Scientific Computing, 38, 5, S101-S122, 2016

[4] J Burghardt, B Leavy, J Guilkey, Z Xue and R Brannon, Application of Uintah-MPM to shaped charge jet penetration of aluminum, IOP Conference Series: Materials Science and Engineering, June 2010

[5] J. D. de St. Germain, J. McCorquodale, S. G. Parker, and C. R. Johnson, Uintah: A massively parallel problem solving environment, in Ninth IEEE International Symposium on High Performance and Distributed Computing, IEEE, Piscataway, NJ, November 2000, pp. 33–41.

[6] Anthony Danalis, Heike Jagode, George Bosilca, Jack Dongarra. PaRSEC in Practice: Optimizing a Legacy Chemistry Application through Distributed Task-Based Execution 3D Digital Imaging and Modeling, International Conference on, 304-313, CLUSTER 2015,IEEE Computer Society, Los Alamitos, CA, USA

[7] Z. DeVito, N. Joubert, F. Palacios, S. Oakley, M. Medina, M. Barrientos, E.Elsen, F. Ham, A. Aiken, K. Duraisamy, E. Darve, J. Alonso, and P. Hanrahan. Liszt: a domain specific language for building portable mesh-based PDE solvers. In Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis (SC '11). ACM, New York, NY, USA

[8] A. Dubey, A. Almgren, John Bell, M. Berzins, S. Brandt, G. Bryan, P. Colella, D. Graves, M. Lijewski, F. Lffler, B. OShea, E. Schnetter, B. Van Straalen, and K. Weide, A survey of high level frameworks in block-structured adaptive mesh refinement packages, Journal of Parallel and Distributed Computing, (2014).

[9] M. N. Farooqi, T. Nguyen, W. Zhang, A.S. Almgren, J. Shalf, and D. Unat. Phase asynchronous AMR execution for productive and performant astrophysical flows. In Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18). IEEE Press, Piscataway, NJ, USA, Article 70, 14 pages, 2018.

[10] J.E. Guilkey, T.B. Harman, and B. Banerjee, An eulerian-lagrangian approach for simulating explosions of energetic devices, Computers and Structures, 85 (2007), pp. 660–674.

[11] J. K. Holmen, A. Humphrey, and M. Berzins. Exploring use of the reserved core. In J. Reinders and J. Jeffers, editors, High Performance Parallelism Pearls, pages 229–242. Elsevier, 2015.

[12] A. Humphrey, D. Sunderland, T. Harman, and M. Berzins. Radiative heat transfer calculation on 16384 gpus using a reverse Monte Carlo ray tracing approach with adaptive mesh refinement. In The 17th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2016), 2016.

[13] A. Humphrey. Scalable Asynchronous Many-Task Runtime Solutions to Globally Coupled Problems. Ph.D Thesis. University of Utah, May 2019.

[14] L.V. Kale, S Krishnan. Charm++: Parallel programming with message-driven objects. In: Wilson, G.V., Lu, P. (eds.) Parallel Programming using C++, pp. 175213. MIT Press (1996)

[15] S. Kumar, A. Humphrey, W. Usher, S. Petruzza, B. Peterson, J. A. Schmidt, D. Harris, B. Isaac, J. Thornock, T. Harman, V. Pascucci, M. Berzins. Scalable Data Management of the Uintah Simulation Framework for Next-Generation Engineering Problems with Radiation, In Supercomputing Frontiers, Springer International Publishing, pp. 219–240. 2018.

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

[17] 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, 26(7):1388–1407, May 2014

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

[19] S. G. Parker, A component-based architecture for parallel multi-physics PDE simulation., Future Generation Computer Systems, 22 (2006), pp. 204–216.

[20] 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 Proceedings of the 5th International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing, WOLFHPC '15, pages 4:1–4:8, New York, NY, USA, 2015. ACM.

[21] B. Peterson, A. Humphrey, J. Holmen T. Harman, M. Berzins, D. Sunderland, H.C. Edwards. Demonstrating GPU Code Portability and Scalability for Radiative Heat Transfer Computations. Journal of Computational Science Volume 27, July 2018, Pages 303-319.

[22] B. Peterson, A. Humphrey, D. Sunderland, J. Sutherland, T. Saad, H. Dasari and M. Berzins, Automatic Halo Management for the Uintah GPU-Heterogeneous Asynchronous Many-Task Runtime. International Journal of Parallel Programming, December 2018.

[23] T. Saad and J.C. Sutherland, Wasatch: An architecture-proof multiphysics development environment using a Domain Specific Language and graph theory. Journal of Computational Science, 17, 639-646.

[24] D. Sahasrabudhe,J. Schmidt and M. Berzins Node Failure Resiliency for Uintah Without Checkpointing. Accepted for Concurrency and Computation: Practice and Experience May 2019.

[25] J.Schmidt and M. Berzins and J. Thornock and T. Saad and J. Sutherland Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre. 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp 458-465, 2013.

[26] Thomas Sterling, Matthew Anderson, Maciej Brodowicz. A Survey: Runtime Software Systems for High Performance Computing Supercomputing Frontiers and Innovations SuperFri.org, 2017.

[27] Vivek Sarkar Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors. Research Monographs in Parallel and Distributed Computing, The MIT Press March 20, 1989, ISBN-10: 0262691302, ISBN-13: 978-0262691307.

[28] J.Spinti, J. Thornock, E. Eddings, P. Smith, and A. Sarofim. Heat transfer to objects in pool fires. In Transport Phenomena in Fires, Southampton, U.K., 2008. WIT Press.

[29] B. Peterson, A. Humphrey, J. Schmidt, and M. Berzins. Addressing Global Data Dependencies in Heterogeneous Asynchronous Runtime Systems on GPUs. In Third International IEEE Workshop on Extreme Scale Programming Models and Middleware, held in conjunction with SC17: The International Conference on High Performance Computing, Networking, Storage and Analysis, 2017. ISBN: 978-3-319-69953-0 DOI: 10.1145/3152041.315208

[30] R. D. Falgout, J. E. Jones, and U. M. Yang. The design and implementation of hypre, a library of parallel high performance preconditioners. In Numerical solution of partial differential equations on parallel computers. Springer, 2006, pp. 267294.

[31] Q. Meng and J. Luitjens and M. Berzins Dynamic Task Scheduling for the Uintah Framework In Proceedings of the 3rd IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS10), 2010, DOI: 10.1109/MTAGS.2010.5699431

[32] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken Legion: expressing locality and independence with logical regions In Proceedings of the International Conference on high performance computing, networking, storage and analysis. IEEE Computer Society Press, 2012, p. 66.

[33] A. Buss, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger STAPL: standard template adaptive parallel library In Proceedngs of the 3rd Annual Haifa Experimental Systems Conference. ACM, 2010, p. 14.

[34] T. Mattson, R. Cledat, Z. Budimlic, V. Cave', S. Chatterjee, B. Seshasayee, R. van der Wijngaart, and V. Sarkar OCR: the open community runtime interface Technical report, 2015.

[35] 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, vol. 23, no. 2, pp. 187198, 2011.

[36] T. Blattner, W. Keyrouz, J. Chalfoun, B. Stivalet, M. Brady, Shujia Zhou A Hybrid CPU-GPU System for Stitching Large Scale Optical Microscopy Images In Parallel Processing (ICPP), 2014 43rd International Conference on , vol., no., pp.1-9, 9-12 Sept. 2014 doi: 10.1109/ICPP.2014.9

[37] Kaiser, Hartmut and Heller, Thomas and Adelstein-Lelbach, Bryce and Serio, Adrian and Fey, Dietmar Hpx: a task based programming model in a global address space In Proc. 8th Int. Conf. Partitioned Global Address Space Programming Models, 2014