SCI INSTITUTE
TECHNICAL REPORT

Uintah+Hedgehog: Combining Parallelism Models for End-to-End Large-Scale Simulation Performance

John K. Holmen, Damodar Sahasrabudhe, Martin Berzins, Alexandre Bardakoff, Timothy J. Blattner, Walid Keyrouz

UUSCI-2021-002

Scientific Computing and Imaging Institute
University of Utah
Salt Lake City, UT 84112 USA

November 22, 2021

Abstract:

The complexity of heterogeneous nodes near and at exascale has increased the need for "heroic" programming efforts. To accommodate this complexity, significant investment is required for codes not yet optimizing for low-level architecture features (e.g., wide vector units) and/or running at large-scale. This paper describes ongoing efforts to combine two codes, Hedgehog and Uintah, lying at both extremes to ease programming efforts. The end goals of this effort are (1) to combine the two codes to make an asynchronous many-task runtime system specializing in both node-level and large-scale performance and (2) to further improve the accessibility of both with portable abstractions. A prototype adopting Hedgehog in Uintah and a prototype extending Hedgehog to support MPI+X hybrid parallelism are discussed. Results achieving ∼60% of NVIDIA V100 GPU peak performance for a distributed DGEMM problem are shown for a naive MPI+Hedgehog implementation before any attempt to optimize for performance.

Authors Note: This is a refereed but unpublished short paper that was submitted to, reviewed for, and accepted in revised form for a presentation of the same material at the Workshop on Hierarchical Parallelism for Exascale Computing (HiPar) held in conjunction with Supercomputing 21.

THE UNIVERSITY OF UTAH

Uintah+Hedgehog: Combining Parallelism Models for End-to-End Large-Scale Simulation Performance

John K. Holmen, Damodar Sahasrabudhe, Martin Berzins
Scientific Computing & Imaging Institute
School of Computing
University of Utah
Salt Lake City, USA

Alexandre Bardakoff, Timothy J. Blattner, Walid Keyrouz
National Institute of Standards & Technology
Gaithersburg, USA

***Abstract*—The complexity of heterogeneous nodes near and at exascale has increased the need for "heroic" programming efforts. To accommodate this complexity, significant investment is required for codes not yet optimizing for low-level architecture features (e.g., wide vector units) and/or running at large-scale. This paper describes ongoing efforts to combine two codes, Hedgehog and Uintah, lying at both extremes to ease programming efforts. The end goals of this effort are (1) to combine the two codes to make an asynchronous many-task runtime system specializing in both node-level and large-scale performance and (2) to further improve the accessibility of both with portable abstractions. A prototype adopting Hedgehog in Uintah and a prototype extending Hedgehog to support MPI+X hybrid parallelism are discussed. Results achieving ∼60% of NVIDIA V100 GPU peak performance for a distributed DGEMM problem are shown for a naive MPI+Hedgehog implementation before any attempt to optimize for performance.**

***Index Terms*—Frameworks, Parallel Architectures, Parallelism and Concurrency, Portability, Software Engineering**

I. Introduction

For codes emphasizing large-scale simulations, "heroic" programming efforts are unfortunately becoming more necessary for current and emerging high performance computing (HPC) systems. For example, nodes to appear in exascale systems pose new challenges relating to increased concurrency, deep memory hierarchies, and heterogeneity (e.g., accommodating large core/thread counts). AI nodes pose similar challenges with, for example, the NVIDIA DGX A100 system featuring two 64-core CPUs and 8 GPUs. These challenges are complicated by the increasing architectural diversity, which may warrant adoption of new programming models for efficient node use. With limited resources, compromises must be made as to where to focus programming efforts.

Two examples of such compromises and design decisions can be seen among Hedgehog and the Uintah Computational Framework. Hedgehog specializes in node-level performance and uses C++ threads and NVIDIA CUDA. Uintah specializes in large-scale simulations and uses an MPI+X hybrid parallelism model supporting a variety of programming models. Both use asynchronous execution to achieve the performance that they specialize in. The combination of the two makes it possible to combine Hedgehog's node-level performance with the multi-node scalability of Uintah.

This paper describes ongoing efforts to combine the two approaches. The end goals of this collaborative effort are (1) to combine Hedgehog and Uintah to make an asynchronous many-task runtime system specializing in both node-level and large-scale performance and (2) further improve the accessibility of both with portable abstractions. Preliminary results show success adopting Hedgehog in Uintah and extending Hedgehog to support MPI+X hybrid parallelism.

The remainder of this paper is structured as follows. Section II provides an overview of Hedgehog. Section III provides an overview of the Uintah Computational Framework. Section V describes two prototypes resulting from this effort and Section VI concludes this paper.

II. Hedgehog

Hedgehog is a general-purpose performance-oriented C++17 headers-only library specializing in maximizing single-node hardware utilization with emphasis on heterogeneous nodes. This is accomplished using a scheduler-free task-based approach that uses data-flow graphs for asynchronous multi-threaded execution at runtime. Hedgehog starts task execution as soon as all the input data are available. For performance portability, Hedgehog implements portable designs for performance [1].

Hedgehog emphasizes node-level performance across wide heterogeneous nodes (e.g., featuring multiple GPUs). For LU decomposition with partial pivoting, Hedgehog performs on par with the LAPACK dgetrf routine compiled with OpenBLAS in multi-threaded mode. For a BLAS-like General Matrix Multiplication routine, Hedgehog achieves >95% of theoretical peak across 4 NVIDIA V100 GPUs, outperforming cuBLASMg and cuBLAS-XT baselines [1]. Unlike Uintah, however, Hedgehog does not yet support multi-node simulations or distributed memory parallelism.

III. The Uintah Computational Framework

The Uintah Computational Framework is an open-source asynchronous many-task (AMT) runtime system specializing in large-scale simulations of fluid-structure interaction problems. These problems are modeled by solving partial differential equations on Cartesian block-structured adaptive mesh refinement grids. This is accomplished through a set of simulation components and libraries based on novel techniques for understanding a broad set of fluid-structure interaction problems [2]. For performance portability, Uintah implements an intermediate performance portability layer that adopts Kokkos indirectly [3].

Uintah emphasizes large-scale simulations across major HPC systems and has been been ported to a diverse set through its lifetime. Recent examples include the NSF Frontera, DOE Lassen [4], NSF Stampede 2 [3], DOE Theta [5], NRCPC Sunway TaihuLight [6], DOE Titan [7], and DOE Mira [2] systems. Good strong-scaling has been shown to 1,728 Intel Knights Landing processors on the NSF Stampede 2 system when using Kokkos::OpenMP [3]. Good strong-scaling has been shown to 1,024 NVIDIA V100 GPUs and 512 IBM POWER9 processors on the DOE Lassen system when using Kokkos::CUDA and Kokkos::OpenMP on the device and host, respectively [4]. Unlike Hedgehog, however, Uintah places little emphasis on optimizing for low-level architecture features as the sheer size of the codebase requires compromises on such development efforts.

IV. Related Approaches

Uintah is one of many asynchronous many-task runtime systems. Other examples include Charm++ [8], DARMA [9], HPX [10], Legion [11], OCR [12], PaRSEC [13], STAPL [14], and StarPU [15]. Hedgehog is one of many options available for developing parallel applications that aim to maximize single-node hardware utilization. Other examples include DPC++ [16], Intel TBB [17], Kokkos [18], OpenACC [19], OpenMP [20], and RAJA [21], SYCL [22]. A review of performance portable programming models for productive exascale computing can be find in a recent survey [23].

Hedgehog is chosen here for the performance of its high performance data-flow model [1] rather than its loop-level parallelism capabilities. Hedgehog's data-flow model is being explored as an alternative to Uintah's per-process infrastructure (e.g., heterogeneous task scheduler [4]). In a similar way to how Uintah offloaded performance portability to Kokkos [3], this is an effort to understand if/how task dependency management can also be offloaded to ease future development efforts. As a part of this, emphasis is placed on understanding shared abstractions and identifying whether or not they can be generalized further (e.g., to help identify key AMT needs).

V. Combining Uintah and Hedgehog

Prototypes discussed here experiment with two communication patterns. The first targets regular communication patterns for stencil-like computations, which are supported by Uintah. The second targets irregular communication patterns for distributed DGEMM, which are not supported by Uintah and require standalone prototypes outside of Uintah. Long-term, the second prototype has the potential to guide the broadening of Uintah's distributed memory parallelism model.

A. Uintah+Hedgehog for Regular Communication

For stencil-like computations, Uintah and Hedgehog can be combined directly. This is made possible by Uintah's ability to automatically generate MPI messages and exchange halo cells surrounding the subdomain for such problems. Specificially, Uintah can be used to manage message passing while Hedgehog is used *locally* within a Uintah task (i.e., a process).

For its simplicity, Uintah's Poisson solver was used to demonstrate how Hedgehog can be integrated into Uintah in such a manner. The resulting prototype required <100 lines of code and consists primarily of two components: (1) a coarse-grained Uintah task and (2) a fine-grained Hedgehog task.

Abbreviated pseudocode for the Uintah task is shown below:

```
1  // Setup the Hedgehog graph and task
2  hh::Graph<outputVars, inputVars> graph(...);
3  auto task = std::make_shared<fineTask>(...);
4
5  // Connect the task to the graph input and output
6  graph.input(task);
7  graph.output(task);
8
9  // Hedgehog graph awaiting inputs
10 graph.executeGraph();
11
12 /*==== Initialize solve ====*/
13 ...
14
15 // Iterate over blocks of data
16 for (int p = 0; p < patches->size(); p++) {
17
18   /*==== Setup boundaries ====*/
19   /*==== Get Uintah data  ====*/
20   ...
21
22   // Assign blocks of data to input
23   auto input = std::make_shared<inputVars>(...);
24
25   // Initiate Hedgehog graph execution
26   graph.pushData(input);
27
28 }
29
30 graph.finishPushingData();
31 graph.waitForTermination();
```

Abbreviated pseudocode for the Hedgehog task is shown below:

```
34 /*==== Poisson solve ====*/
35 ...
36
37 // Assign blocks of data to output
38 auto result = std::make_shared<outputVars>(...);
39
40 this->addResult(result);
```

Broadly, this prototype translates Uintah's existing Poisson solver into a Hedgehog task. More specifically, a coarse-grained Uintah task is created and used to declare and instantiate the Hedgehog data-flow graph (line 2), which uses

a fine-grained Hedgehog task to implement Uintah's Poisson solver. At run-time, the coarse-grained Uintah task is executed to setup (lines 5-11) and execute (line 14) the Hedgehog data-flow graph, which will execute the former Uintah Poisson solver tasks as fine-grained Hedgehog tasks.

This approach is currently limited to using Uintah's MPI-only task schedulers. As a result, use of per MPI process resources (e.g., cores, threads) is managed by Hedgehog. Long-term, Uintah's heterogeneous MPI+PPL task scheduler [4] could be used as a starting point for further improving node performance. Such a scheduler would allow for a balance to be struck between Uintah's overdecomposition of tasks and Hedgehog's overdecomposition of data when optimizing run configurations. Note, performance comparisons are not included here as the API is subject to change as part of long-term goals aiming to generalize both Uintah and Hedgehog API. In general, Hedgehog overheads are on the order of microseconds [1].

B. MPI+Hedgehog for Irregular Communication

The second prototype is for problems where the communication pattern is not supported by Uintah. To support such patterns, separate Hedgehog tasks are created to handle communication and Hedgehog is extended to support MPI+X hybrid parallelism. Use of Hedgehog's data-flow execution model ensures that communication is completed before computation and allows for communication tasks to overlap computation tasks.

The resulting prototype uses Cannon's algorithm [24] for distributed double-precision general matrix-matrix multiplication (distributed DGEMM) $C = A \times B$ on GPUs. This algorithm arranges MPI processes in a two-dimensional grid, decomposes matrices A and B among MPI processes, and rotates blocks horizontally/vertically in each iteration. Each process multiplies the blocks it owns and accumulates results into a destination block of C, which remains constant for each rank. The communication pattern is designed such that each rank gets blocks of A and B corresponding to the destination block of C for each iteration of the algorithm. Such a circular communication pattern of passing entire blocks is not yet supported by Uintah and requires a standalone prototype. Long-term, the aim is to integrate Hedgehog "communication tasks" in Uintah's task scheduler to accomodate irregular commmunication patterns.

This prototype is based on Hedgehog's Tutorial 4 [25], which implements DGEMM on a single GPU using a single process. Tutorial 4 is extended to support "distributed DGEMM" using Cannon's algorithm with key changes including: (1) newly implemented InitComm and FinalizeComm tasks to send and receive blocks needed for the next iteration of Cannon's algorithm, (2) updates to the cudaAdditionTask to accumulate the result matrix on GPUs instead of CPUs as done in Tutorial 4, and (3) all the CUDA tasks are executed asynchronously.

Preliminary experiments show distributed DGEMM performance of 18.8 TFLOPS across four NVIDIA V100 GPUs for matrices of size 32,768 x 32,768. These numbers translate into 4.7 TFLOPS per GPU which is $\sim$60% of the theoretical peak performance for double-precision. This result is encouraging as this was a naive MPI+Hedgehog implementation before any attempt to optimize for performance. Experiments were run using 1 MPI process per GPU with each MPI process using 8 threads for the product tasks and 1 thread for all other tasks.

C. Moving Forward

Aside from communication patterns, the key difference between prototypes is timestepping, which is a key characteristic of typical Uintah problems. For this reason, Uintah's task schedulers have been designed for timestepped problems that communicate halo data. Implementation of a more general Uintah task scheduler is another long-term goal of this effort. Such a scheduler would allow Uintah to support a wider range of problems and more general communication patterns. A natural next step for accomplishing this is to implement the distributed DGEMM problem in Uintah and use it to guide scheduler developments.

VI. Conclusions and Future Work

This work has helped improve our understanding of how Hedgehog's shared memory parallelism model interoperates with Uintah's MPI+X hybrid parallelism model. Perhaps more important, successful prototypes suggest a potential to successfully combine Hedgehog's node-level performance with the multi-node scalability of Uintah. Such a combination promotes more productive use of current and emering HPC systems.

The Uintah+Hedgehog and MPI+Hedgehog prototypes discussed here offer encouragement as we prepare to more tightly couple Hedgehog and Uintah. Immediate next steps include further optimization of the distributed DGEMM prototype. Long-term next steps include implementing a Uintah task scheduler supporting Hedgehog tasks. As part of this effort, emphasis will be placed on generalizing portable abstractions used by each to simplify API for end users.

Acknowledgment

This material is based upon work supported by the National Institute of Standards and Technology under Award Number(s) 70NANB20H018. This research used resources of the National Institute of Standards and Technology. Support for J. K. Holmen and D. Sahasrabudhe also comes from the University of Texas at Austin under Award Number(s) UTA19-001215 and a gift from the Intel Parallel Computing Centers Program.

Disclaimer

REFERENCES

[1] A. Bardakoff, B. Bachelet, T. Blattner, W. Keyrouz, G. C. Kroiz, and L. Yon, "Hedgehog: Understandable scheduler-free heterogeneous asynchronous multithreaded data-flow graphs," in *2020 IEEE/ACM 3rd Annual Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*, 2020, pp. 1–15.

[2] 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," *SIAM Journal on Scientific Computing*, vol. 38, no. 5, pp. 101–122, 2016.

[3] J. K. Holmen, B. Peterson, and M. Berzins, "An approach for indirectly adopting a performance portability layer in large legacy codes," in *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 2019, pp. 36–49.

[4] J. K. Holmen, D. Sahasrabudhe, and M. Berzins, "A heterogeneous MPI+PPL task scheduling approach for asynchronous many-task runtime systems," in *Proceedings of the Practice and Experience in Advanced Research Computing 2021 on Sustainability, Success and Impact (PEARC21)*. ACM, 2021.

[5] B. Peterson, A. Humphrey, J. K. Holmen, T. Harman, M. Berzins, D. Sunderland, and H. C. Edwards, "Demonstrating GPU code portability and scalability for radiative heat transfer computations," *Journal of Computational Science*, vol. 27, pp. 303 – 319, 2018.

[6] Z. Yang, D. Sahasrabudhe, A. Humphrey, and M. Berzins, "A preliminary port and evaluation of the Uintah AMT runtime on Sunway TaihuLight," in *9th IEEE International Workshop on Parallel and Distributed Scientific and Engineering Computing (PDSEC 2018)*. IEEE, May 2018.

[7] 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 *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 1222–1231.

[8] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. New York NY USA: ACM, 1993, pp. 91–108.

[9] J. Wilke, D. Hollman, N. Slattengren, J. Lifflander, H. Kolla, F. Rizzi, K. Teranishi, and J. Bennett, "Darma 0.3. 0-alpha specification," Sandia National Laboratory (SNL-CA), Livermore, CA, USA, Tech. Rep., 2016.

[10] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York NY USA: ACM, 2014, pp. 6:1–6:11.

[11] 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.

[12] T. G. Mattson, R. Cledat, V. Cavé, V. Sarkar, Z. Budimlić, S. Chatterjee, J. Fryman, I. Ganev, R. Knauerhase, and M. Lee, "The open community runtime: a runtime system for extreme scale computing," in *2016 IEEE High Performance Extreme Computing Conf. (HPEC)*. IEEE, Sept. 2016, pp. 1–7.

[13] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Herault, and J. J. Dongarra, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science Engineering*, vol. 15, no. 6, pp. 36–45, Nov 2013.

[14] 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 *Proc. 3rd Annu. Haifa Experimental Systems Conf.* ACM, 2010, p. 14.

[15] 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. 187–198, 2011.

[16] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[17] A. Kukanov and M. J. Voss, "The foundations for scalable multi-core software in intel threading building blocks." *Intel Technology Journal*, vol. 11, no. 4, 2007.

[18] C. Trott, D. Lebrun-Grandie, D. Arndt, J. Ciesko, V. Dang, N. Ellingwood, R. Gayatri, E. Harvey, D. S. Hollman, D. A. Ibanez *et al.*, "Kokkos 3: Programming model extensions for the exascale era," *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[19] R. Farber, *Parallel programming with OpenACC*. Newnes, 2016.

[20] R. Chandra, L. Dagum, D. Kohr, R. Menon, D. Maydan, and J. McDonald, *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[21] R. D. Hornung and J. A. Keasler, "The raja portability layer: overview and status," Lawrence Livermore National Laboratory (LLNL), Livermore, CA, Tech. Rep., 2014.

[22] R. Keryell, M. Rovatsou, and L. Howes, "Khronos Group SYCL 1.2.1 Specification," https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf, 2019.

[23] A. Johnson, "Area exam: General-purpose performance portable programming models for productive exascale computing," 2020.

[24] L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*. Montana State University, 1969.

[25] "Hedgehog Tutorials," https://pages.nist.gov/hedgehog-Tutorials/.