Optimizing the Hypre solver for manycore and GPU architectures

Damodar Sahasrabudhe[a], Rohit Zambre[b], Aparna Chandramowlishwaran[b], Martin Berzins[a]

[a] *SCI Institute, University of Utah, Salt Lake City*
[b] *EECS, University of California, Irvine*

Abstract

The solution of large-scale combustion problems with codes such as Uintah on modern computer architectures requires the use of multithreading and GPUs to achieve performance. Uintah uses a low-Mach number approximation that requires iteratively solving a large system of linear equations. The Hypre iterative solver has solved such systems in a scalable way for Uintah, but the use of OpenMP with Hypre leads to at least $2\times$ slowdown due to OpenMP overheads. The proposed solution uses the MPI Endpoints within Hypre, where each team of threads acts as a different MPI rank. This approach minimizes OpenMP synchronization overhead and performs as fast or (up to $1.44\times$) faster than Hypre's MPI-only version, and allows the rest of Uintah to be optimized using OpenMP. The profiling of the GPU version of Hypre shows the bottleneck to be the launch overhead of thousands of micro-kernels. The GPU performance was improved by fusing these micro-kernels and was further optimized by using Cuda-aware MPI, resulting in an overall speedup of 1.16–$1.44\times$ compared to the baseline GPU implementation.

The above optimization strategies were published in the International Conference on Computational Science 2020 [1]. This work extends the previously published re-

*The authors thank Department of Energy, National Nuclear Security Administration (under Award Number(s) DE-NA0002375) and Intel Parallel Computing Center, for funding this work. This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357 and also of Lawrence Livermore National Laboratory. J. Schmidt, J. Holmen, A. Humphrey and the Hypre team are thanked for the help.

*Corresponding author
Email address: damodars@sci.utah.edu (Damodar Sahasrabudhe)

search by carrying out the second phase of *communication-centered optimizations* in Hypre to improve its *scalability* on large-scale supercomputers. This includes an efficient non-blocking inter-thread communication scheme, communication-reducing patch assignment, and expression of logical communication parallelism to a new version of the MPICH library that utilizes the underlying network parallelism [2]. The above optimizations avoid communication bottlenecks previously observed during strong scaling and improve performance by up to 2× on 256 nodes of Intel Knight's Landing processor.

Keywords: Hypre, MPI EndPoints, Multithreading, Manycore processors, GPUs, Performance Optimizations

---

1. Introduction

The asynchronous many task Uintah Computational Framework [3] solves complex large-scale partial differential equations (PDEs) involved in multi-physics problems such as combustion and fluid interactions. One of the important tasks in the solution of many such large scale PDE problems is to solve a system of linear equations. Examples are the linear solvers used in the solution of low-Mach-number combustion problems or incompressible flow. Uintah-based simulations of next-generation combustion problems have been successfully ported to different architectures, including heterogeneous architectures and, have scaled up to 96k, 262k, and 512k cores on the NSF Stampede, DOE Titan, and DOE Mira supercomputers respectively [3]. Such simulation employs the Arches component of Uintah. Arches is a three dimensional, Large Eddy Simulation (LES) code developed at the University of Utah. Arches simulates heat, mass, and momentum transport in reacting flows by using a low Mach number $(Ma < 0.3)$ variable density formulation [4]. The solution of a pressure projection equation at every time sub-step is required for the low-Mach-number pressure formulation. This is done using the Hypre package [4]. Hypre supports different iterative and multigrid methods, has a long history of scaling well [5, 6] and has successfully weak scaled up to 500k cores when used with Uintah [7].

While past Uintah simulations were carried out [3] on DOE Mira and Titan sys-

tems [7], the next-generation of simulations will be run on manycore processors (such as DOE's Theta and NSF's Frontera) and GPU architectures (such as DOE's Lassen, Summit, and Aurora). On both classes of machines, the challenge for library software is then to move away from an MPI-only approach in which one MPI process runs per core to a more efficient approach in terms of storage and execution models. On many-core processors, a common approach is to use a combination of MPI and OpenMP to exploit the massive parallelism. In the case of GPUs, the OpenMP parallel region can be offloaded to a GPU with CUDA or OpenMP 4.5. It is also possible to use portability layers such as Kokkos [8] to automate the process of using either OpenMP or CUDA. The MPI-only configuration for Uintah spawns one single-threaded rank per core and assigns one patch per rank. In contrast, the Uintah's Unified Task Scheduler was developed to leverage multithreading and also to support GPUs [9]. Work is in progress to implement portable multithreaded Kokkos-OpenMP and Kokkos-CUDA [8] based schedulers and tasks to make Uintah portable for future heterogeneous architectures. These new Uintah schedulers are based on *teams of threads*. Each rank is assigned multiple patches, which are distributed among thread teams. Teams of threads then process the patches in parallel (task parallelism) while threads within a team, work on a single patch (data parallelism). This hybrid design has proven useful on manycore systems and in conjunction with Kokkos has led to dramatic improvements in performance [8].

The challenge of realizing similar performance improvements with Uintah's use of Hypre and its Structured Grid Interface (Struct) is addressed in this work, so that Hypre performs as well (or better) in a threaded environment as in the MPI case. Hypre's structured multigrid solver, PFMG [10], is designed to be used with unions of logically rectangular sub-grids and is a semi-coarsening multigrid method for solving scalar diffusion equations on logically rectangular grids discretized with up to 9-point stencils in 2D and up to 27-point stencils in 3D. Baker et al. [10] report that various versions of PFMG are between 2.5–7$\times$ faster than the equivalent algebraic multigrid (AMG) options inside Hypre because they are able to take account of the grid structure. When Hypre is used with Uintah, the linear solver algorithm uses the Conjugate Gradient (CG) method with the PFMG preconditioner based upon a Jacobi relaxation method

inside the structured multigrid approach [4].

The equation (1) that is solved in Uintah is derived from the numerical solution of the Navier-Stokes equations and is a Poisson equation for the pressure, p, whose solution requires the use of a solver such as Hypre for large sparse systems of equations. While the form of (1) is straightforward, a large number of variables, for example, 6.4 billion in [4], represents a challenge that requires large scale parallelism. One key challenge with Hypre is that only one thread per MPI rank can call Hypre. This forces Uintah to join all the threads and teams before Hypre can be called, after which the main thread calls Hypre. Internally Hypre uses all the OpenMP threads to process cells within a domain, while patches are processed serially. From the experiments reported here, it is this particular combination that introduces extra overhead and causes the observed performance degradation. Thus, the challenge is to achieve performance with the multithreaded and GPU versions of Hypre but without degrading the optimized performance of the rest of the code.

$$\nabla^2 p = \nabla \cdot \mathbf{F} + \frac{\partial^2 \rho}{\partial t^2} \equiv R \tag{1}$$

1.1. Enabling Hypre on New Architectures

In moving Hypre to manycore architectures, OpenMP was introduced to support multithreading [11]. However, in contrast to the results in [11], a dramatic slowdown of 3–8$\times$ was observed when using Hypre with Uintah in an OpenMP multithreaded environment compared to the MPI-only version. Baker et al. make similar observations using a test problem with PFMG solver and up to 64 patches per rank. They observe a slowdown of 8–10$\times$ between the MPI-only and MPI+OpenMP versions [6]. The challenges of OpenMP in Hypre forces Uintah to either single-threaded (MPI-only) version of Hypre or to use OpenMP with one patch per rank. This defeats the purpose of using OpenMP.

This work shows that the root cause of the slowdown is the use of OpenMP pragmas at the innermost level of the loop structure. However, the straightforward solution of moving these OpenMP pragmas to a higher loop level does not offer the needed performance. The solution adopted here is to use an alternate threading model using *MPI scalable Endpoints* [12, 13] to solve the slowdown problem and to achieve a speedup

consistent with the results observed by [11, 6]. This approach requires overriding MPI calls to simulate MPI endpoints behavior. The current MPI standard assigns an MPI rank to a process, and all threads within a process share the same rank. The proposed endpoints approach allows assigning an MPI rank to an *EndPoint* (EP), where an EP can be a thread, a team of threads, or a process [14]. Thus, each thread (or a team of threads) attached to an EP acts as an individual MPI rank. There can be multiple EPs with different MPI ranks within a process, and each endpoint can execute its computations and independently communicate with other EPs using MPI messages. Additional optimizations such as vectorization, funneled communication , and a lightweight threading model further improves the performance. In this extension of prior work, three *communication-centered optimizations* are introduced: an efficient inter-thread communication scheme, communication-reducing patch assignment, and utilization of network parallelism through a state-of-the-art MPICH library capable of mapping logically parallel MPI communication to distinct network contexts. The new enhancements improve the scalability of Hypre and result in an overall speedup of 1.7–2.4× over the MPI-only version.

In optimizing Hypre performance for GPUs, Hypre 2.15.0 was chosen as a baseline on Nvidia V100 GPUs, to characterize the performance. Profiling on GPU reveals the launch overhead of GPU kernels to be the primary bottleneck and occurs because of launching thousands of *micro* kernels. The problem is fixed by fusing these micro kernels and using GPU's constant cache memory. Finally, Hypre is modified to leverage CUDA-aware MPI on the Lassen cluster which gives an extra 10% improvement.

The main contributions of this work are to:

- Introduce the MPI EP model in Hypre (called Hypre-EP) to avoid the performance bottlenecks observed with OpenMP. This can enable better overall performance in the future when running the full simulation using a multithreaded task scheduler within Uintah AMT.

- Identify the bottlenecks in Hypre-EP and improve its performance and scalability beyond the MPI-only version by adding new communication-centered optimizations to get the overall speedup of 1.7–2.4× over the MPI-only version.

- Optimize the CUDA version of Hypre to improve CPU to GPU speedups ranging from 2.3× to 4× in the baseline version to the range of 3× to 6× in the optimized version. This enables large-scale combustion simulations on current and future GPU based supercomputers.

2. CPU Performance Analysis and Optimizations: Phase I

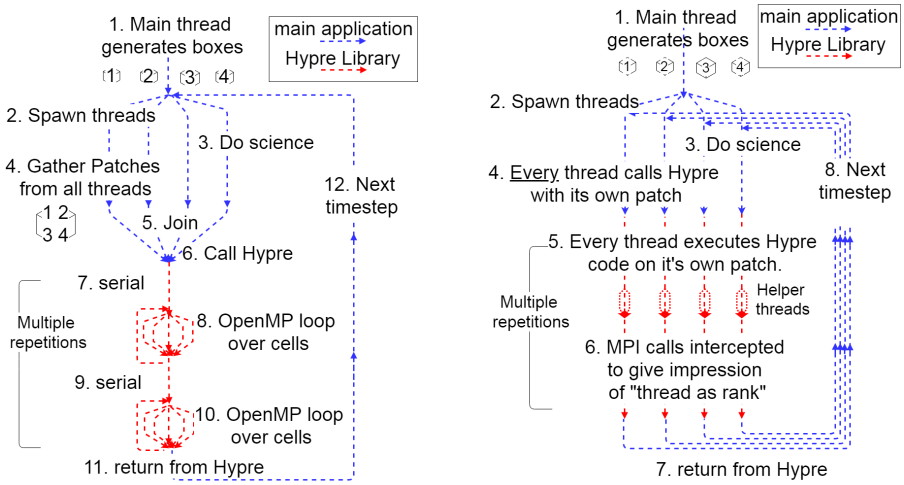
This section analyzes the performance challenges and current limitations of Hypre with OpenMP. Then, the MPI EndPoints approach is explained, followed by experimental evaluation on a large-scale system.

2.1. Performance Analysis of OpenMP

To understand the slowdown of Hypre with OpenMP, the PFMG preconditioner and the PCG solver is profiled with a standalone code that solves a 3D Laplace equation on a regular mesh, using a 7 point stencil. The solve step is the computational core since it runs iteratively while the setup is executed only once. This representative example mimicks the use of Hypre in Uintah where each MPI rank derives its patches (Hypre boxes) based on its rank and allocated the required data structures accordingly. Each rank owns a minimum of 4 patches to a maximum of 128 patches where each patch is initialized by its rank owner. Intel's Vtune amplifier and `gprof` are used for profiling on a KNL node with 64 cores. The MPI Only version executes with 64 ranks (where 1 rank is assigned to each core), and the MPI + OpenMP version runs 1x64, 2x32, 4x16, 8x8, and 16x4 ranks and threads, respectively.

The Struct interface of Hypre is called – first to carry on the setup and then to solve the equations. The solve step is repeated 10 times to simulate timesteps in Uintah. Each test problem uses a different combination of domain and patch sizes: a 64^3 or 128^3 domain is used with 4^3 patches of sizes 16^3 or 32^3. A 128^3 or 256^3 domain is used with 8^3 patches of sizes 16^3 or 32^3. Multiple combinations of MPI ranks, number of OpenMP threads per rank, and patches per rank is explored and compared against the MPI Only version. Each solve step takes about 10 iterations to converge on average.

Two main performance bottlenecks observed during profiling are discussed below:



(a) Existing Control Flow of Hypre

(b) New Control Flow of Hypre

Figure 1: Software Design of Hypre

1. **OpenMP synchronization overhead and insufficient work**. Figure 1a shows the code structure of how an application (Uintah) calls Hypre. Uintah spawns its threads, generates patches, and executes tasks scheduled on these patches. When Uintah encounters the Hypre task, all threads join and the main thread calls Hypre. Hypre then spawns its own OpenMP threads and continues. With 4 MPI ranks and 16 OpenMP threads in each, Vtune reports a Hypre solve of 595 seconds. Of this time, the OpenMP overhead accounts for 479 seconds, and spin time is 12 seconds. The PFMG-CG algorithm calls 1000s of *micro-kernels* during the solve step. Each micro kernel performs operations such as matrix-vector multiplication, scalar multiplication, relaxation, etc. and uses OpenMP to parallelize over the patch cells. The *multi-grid* nature of the PFMG algorithm causes the coarsening of the mesh at every level. The number of cells reduces from n^3 to 1 and again increases back to n^3 with refining. Such a reduction in the number of cells makes the kernels extremely lightweight. The light workload is not enough to offset the overhead of the OpenMP thread barrier at the end of every parallel for and results in 6$\times$ performance degradation. Moreover, the OpenMP overhead grows with the number of OpenMP threads per rank and patches per rank. As a result, Hypre does not benefit from multiple threads and cores.

2. **Failure of auto-vectorization**. Hypre uses *loop iterator* macros (e.g., BoxLoop) which expand into multidimensional for loops. These iterator macros use a dynamic stride passed as an argument. Although the dynamic stride is necessary for some use cases, many use cases have a fixed unique stride. As the compiler cannot determine the dynamic stride a priori, the loop is not auto-vectorized.

2.2. Restructuring OpenMP Loops and MPI Endpoints

A straightforward solution to the bottlenecks identified above is to parallelize the outermost loop, namely the loop at the *patch* level. This approach is evaluated for the Hypre function `hypre_PointRelax`. Table 1 shows execution times for the MPI Only, default hybrid MPI + OpenMP (with OpenMP pragmas around cell loops), and the new hybrid MPI + OpenMP implementations. In the new hybrid MPI + OpenMP code, parallelization is over mesh patches instead of cells and each thread processes one or more mesh patches.

Table 1: Comparison of MPI vs. OpenMP execution times using 64 32^3 mesh patches.

Hypre Configuration	**Execution time (s)**
MPI Only 64 ranks	1.45
Default hybrid: 4 ranks each with 16 threads, OpenMP over cells	5.61
New hybrid: 4 ranks each with 16 threads, OpenMP over patches	3.19
MPI Endpoints: 4 ranks each with 4 teams each with 4 threads	1.56

The parallelization over mesh patches improves the performance by 1.75×. Nonetheless, this is still 2× slower than the MPI Only version.

Adopting the MPI Endpoints (EP) approach, illustrated in Figure 1b, can bridge this performance gap. In this new approach, each Uintah's *team of threads* acts independently as if it is a separate rank and calls Hypre, passing its patches. Each team processes its patches and communicates with other real and virtual ranks (*virtual rank = real rank × number of teams + team id*). MPI wrappers convert virtual ranks to real

ranks and vice versa during MPI communication. This conversion generates an impression of each team being an MPI rank and the behavior is similar to the MPI Only implementation. The smaller team size (compared to the entire rank) minimizes overhead incurred in fork-joins in the existing OpenMP implementation, yet can exploit the abundant data parallelism available on manycore processors.

Nonetheless, the design and implementation of MPI EP are not without its challenges which are outlined below.

(a) **Race Conditions**: All global and static variables are converted to thread_local variables to avoid race conditions.

(b) **MPI Conflicts**: A potentially challenging problem is to avoid MPI conflicts due to threads. In Hypre, only the main thread is designed to handle all MPI communications. With the MPI EP approach, each team makes its MPI calls. As Hypre already has MPI wrappers in place for all MPI functions, adding logic in every wrapper function to convert between a virtual rank and a real rank and to synchronize teams during MPI reductions is sufficient to avoid MPI conflicts.

(c) **Locks within MPI**: The MPICH implementation which is the base for Intel MPI and Cray MPI uses global locks. As a result, only one thread can be inside the MPI library for most of the MPI functions. This can be a limitation for the new approach as the number of threads per rank is increased. To overcome this, an extra thread is spawned called the *communication thread* and all the communication is funneled through this thread during the solve phase. This provides minimum thread wait times and results in the highest throughput.

2.3. Optimizations in Hypre: Phase I

The three key optimizations in the proposed approach are:

MPI Endpoint: A dynamic conversion mechanism between the virtual and the real rank along with encoding of source and destination team ids within the MPI message tag simulates MPI Endpoint behavior. Also, MPI reduce and probe calls need extra processing. These changes are described below.

```
int g_num_teams;
__thread int tl_team_id;
int hypre_MPI_Comm_rank( MPI_Comm comm, int *rank ){
  int mpi_rank, ierr;
  ierr = MPI_Comm_rank(comm, &mpi_rank);
  *rank = mpi_rank * g_num_teams + tl_team_id;
  return ierr;
}
```

Figure 2: Pseudo code of MPI EP wrapper for `MPI_Comm_rank`.

(a) **`MPI_Comm_rank`**: This command is mapped by using the formula above relating ranks and teams. Figure 2 shows the pseudo-code to convert the real MPI rank to the virtual MPI EP rank using the formula "mpi_rank $\times$ g_num_teams + tl_team_id". The global variable `g_num_teams` and the thread-local variable `tl_team_id` are initialized to the number of teams and the team id. Thus each endpoint gets an impression of a standalone MPI rank. Similar mapping is used in the subsequent wrappers.

(b) **`MPI_Send, Isend, Recv, Irecv`**: The source and destination team ids are encoded in the tag values. The real rank and the team id are easily computed from the virtual rank by dividing by the number of teams.

(c) **`MPI_Allreduce`**: All teams within a rank carry out a local reduction first and then only the zeroth thread calls the `MPI_Allreduce` collective passing the locally reduced buffer as an input. Once the `MPI_Allreduce` returns, all teams copy the data from the globally reduced buffer back to their output buffers. Thread synchronization is achieved using lock-free busy waiting using C11 atomic primitives.

(d) **`MPI_Iprobe and Improbe`**: Each team is assigned a message queue internally. Whenever a probe is executed by any team, it first checks its internal queue for the message. If the handle is found, it is retrieved using `MPI_mecv`. If the handle is not found in the queue, then the `Improbe` function is issued and if

the message at the head of the MPI queue is destined for the same team, then `MPI_mecv` is again issued. If the incoming message is tagged for another team, then the receiving team inserts the handle in the destination team's queue. This method avoids the blocking of MPI queues when the intended recipient of the MPI queue's head is busy and does not issue a probe.

(e) **`MPI_GetCount`**: In this case, the wrapper simply updates source and tag values.

(f) **`MPI_Waitall`**: The use of a global lock in MPICH `MPI_Waitall` stalls other threads and MPI operations do not progress. Hence a `MPI_Waitall` wrapper is implemented by calling `MPI_Testtall` and busy waiting until `MPI_Testtall` returns true. This results in 15-20% speedup over threaded `MPI_Waitall`.

Improving auto-vectorization: The loop iterator macros in Hypre operate using dynamic stride which prevents the compiler from vectorizing these loops. To overcome this limitation, additional macros are introduced specifically for the unit stride cases. The compiler is then able to auto-vectorize some of the loops and results in an additional 10 to 20% performance improvement depending on the patch size.

Interface for `parallel_for` and lightweight Hierarchical Parallelism: A downside of explicitly using OpenMP in Hypre is possible incompatibilities with other threading models. In the spirit of [8], an interface is introduced that allows users to pass their version of `parallel_for` as a function pointer during initialization and this user-defined `parallel_for` is called by simplified BoxLoop macros. Therefore, Hypre users can implement `parallel_for` in any threading model. The new interface allows a lightweight implementation of a threading model, in which all the threads, including the worker threads, are spawned at the beginning of the execution. The main thread of every team acts as an MPI Endpoint, while worker threads do *busy waiting* until needed. When the main thread calls a `parallel_for`, it shares the C++11 lambda and iteration count with the worker threads who execute the lambda in parallel. This approach is similar to OpenMP in principle but uses the atomic primitives and busy-waiting to achieve lock-free thread synchronization. As a result, the lightweight

`parallel_for` performs faster than a `for` loop parallelized using `#pragma omp parallel for`, which typically uses pthread-based locks and conditional variables.

Table 2: OpenMP vs. Custom parallel_for on KNL: Execution times in seconds

Patch Size	$\mathbf{16^3}$	$\mathbf{32^3}$	$\mathbf{64^3}$
OpenMP	1.1	1.5	3.2
Custom `parallel_for`	0.28	0.56	2.1
MPI Only	0.15	0.5	2.8

Table 2 shows the solve time per timestep in seconds for a problem with 64 patches of size 64^3 on a KNL node. The MPI Only version is run using 64 ranks with one patch per rank. The OpenMP and custom `parallel_for` versions are run using four ranks with four teams per rank and four worker threads per team. Each team gets four patches. Each rank spawns an extra communication thread. The only difference between the two threaded versions is the threading model. The lightweight `parallel_for` model runs 3.9×, 2.7×, and 1.5× times faster than the OpenMP version for the patch sizes 16^3, 32^3, and 64^3, respectively. Increasing the patch size reduces the performance gap between the two versions because the extra workload compensates for the OpenMP overhead. The lightweight `parallel_for` performs in a comparable way to the MPI Only version for a 32^3 patch and results in a speedup of 2.7× over the OpenMP-based `parallel_for`. The performance improvement over Hypre MPI Only is due to vectorization and reduced MPI communication. The customized lightweight `parallel_for` makes these improvements apparent through lower overheads than the OpenMP-based `parallel_for`.

The main advantage of using Hierarchical Parallelism is to reduce the number of EPs compared to pure EP based execution where each thread acts as an EP. Reducing the number of EPs reduces MPI communication, both point-to-point (such as `MPI_send-receive`) and collective communication (such as `MPI_Allreduce`). MPI EP with hierarchical parallelism can minimize overheads than either using all threads as EPs, which wastes a lot of time in `MPI_waitall` or using all threads as worker threads, which causes huge synchronization overheads (as in the default Hypre-

OpenMP). As a result, the combination of EP and hierarchical parallelism can result in the best performance.

2.4. Experimental Setup on Theta

Choosing the patch size: Initial experiments using only the Hypre solve component on small node counts shows that the performance improves with the patch size. Table 3 compares both hybrid MPI+OpenMP and MPI EP implementations against the MPI Only for different patch sizes. MPI+OpenMP always performs slower than the MPI Only version, although the performance improves marginally as the patch size increases. On the other hand, the MPI EP model performs nearly as well as the MPI Only version for 16^3 and 32^3 patch sizes on 2 and 4 nodes but breaks down at the end of scaling. With 64^3 patches, however, MPI EP performed up to $1.4\times$ faster than the MPI Only version. As a result, the patch size of 64^3 is chosen for the scaling experiments on the representative problem. These results carry across to the larger node counts. Strong scaling studies with 16^3 patches show the MPI+OpenMP approach performs $4\times$ to $8\times$ slower than the MPI Only version. In the case of Hypre-MPI EP, the worst-case slowdown of $1.8\times$ is observed for 512 nodes and the fastest execution matched the time of Hypre-MPI Only. This experience together with the results presented above stresses the importance of using larger patch sizes, 64^3 and above, to achieve scalability and performance.

Table 3: Speedups of the MPI+OpenMP and MPI EP versions compared to the MPI Only version for different patch sizes.

Patch size:	**16^3**		**32^3**		**64^3**	
Nodes	**MPI+ OpenMP**	**MPI EP**	**MPI+ OpenMP**	**MPI EP**	**MPI+ OpenMP**	**MPI EP**
2	0.2	0.9	0.2	1.2	0.5	1.4
4	0.2	0.8	0.2	0.9	0.4	1.4
8	0.2	0.5	0.3	0.6	0.5	1.3

As the process of converting Uintah's legacy code to Kokkos based portable code that can use either OpenMP or CUDA is still in progress, not all sections of the code can

be run efficiently in the multi-threaded environment. Hence a representative problem containing the two most time-consuming components is chosen for the scaling studies on DOE Theta. The two main components are: (i) Reverse Monte Carlo Ray Tracing (RMCRT) that solves for the radiative-flux divergence during combustion [15] and (ii) pressure solve which uses Hypre. RMCRT has previously been ported to utilize a multi-threaded approach that performs faster than the MPI Only version and also reduces memory utilization [16]. The second component, Hypre solver, is optimized as part of this work for a multi-threaded environment. The combination of these two components shows the impact of using an efficient implementation of multi-threaded Hypre code on the overall simulation of combustion.

Three different mesh sizes are chosen for strong scaling experiments on DOE Theta: small (512^3), medium (1024^3) and large (2048^3). The coarser mesh for RMCRT is fixed at 128^3. Each node of DOE Theta contains one Intel's Knights Landing (KNL) processor with 64 cores per node, 16 GB of the high bandwidth memory (MCDRAM) and AVX512 vector support. The MCDRAM is configured in a cache-quadrant mode for the experiments in this paper. Hypre and Uintah are compiled using Intel Parallel Studio 19.0.5.281 with Cray's MPI wrappers and compiler flags "-std=c++11 -fp-model precise -g -O2 -xMIC-AVX512 -fPIC". One MPI process is launched per core (i.e., 64 ranks per node) while running the MPI Only version. For the MPI+OpenMP and MPI EP versions, four ranks are launched per node (one per KNL quadrant) with 16 OpenMP threads per rank. The flexibility of choosing team sizes in MPI EP allows running the multiple combinations of teams $\times$ worker threads within a rank: 16×1, 8×2, and 4×4. The best performing combinations among these are selected.

2.5. Results and Evaluation on Theta

Figure 3a shows the execution time per timestep in seconds for the RMCRT component on DOE Theta. The multi-threaded execution of RMCRT shows improvements between $2\times$ to $2.5\times$ over the MPI Only version for the small problem and $1.4\times$ to $1.9\times$ for the medium size problem. Furthermore, the RMCRT speedups increase with the scaling. This performance boost is due to the all-to-all communication in the RMCRT algorithm that is reduced by $16\times$ when using 16 threads per rank. The multi-

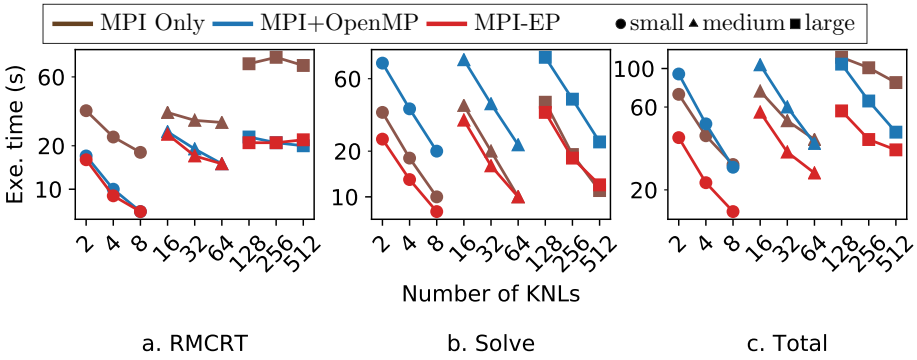


Figure 3: Theta results: The execution time/timestep in seconds for RMCRT, Hypre and total time.

Table 4: Theta results: Communication wait time for MPI EP.

Nodes	2	4	8	16	32	64	128	256	512
MPI Wait	2.4	1.4	1.7	6	3.9	5	11	7.5	6
Solve	24	13	8	32	16	10	36	18	12
% Comm	10%	11%	21%	19%	24%	50%	30%	42%	50%

threaded version also results in up to 4× less memory allocation per node. However, the RMCRT performance improvements are hidden by the poor performance of Hypre in the MPI+OpenMP version. Compared to the MPI Only version, a slowdown of 2× is observed in Hypre MPI+OpenMP despite using 64^3 patches (Figure 3b). The slowdowns observed are as worse as 8× for smaller patch sizes. Using an optimized version of Hypre (MPI EP + partial vectorization) not only avoids these slowdowns but also provides speedups from 1.16-1.44× over the MPI Only solve. The only exceptions are 64, 256, and 512 nodes, where there is no extra speedup for Hypre because the scaling breaks down. Because of the faster computation times (Figure 3b), lesser time is available for the MPI EP model to effectively hide the communication and also wait time due to locks within MPI starts dominating. Table 4 shows the percentage of solve time spent waiting for communication. During the first two steps of scaling, the communication wait time also scales, but increases during the last step for 8 and 64 nodes. The MPI wait time increases from 24% for 32 nodes to 50% for 64 nodes

and the communication starts dominating the computation because there is not enough work per node.

As both components take advantage of the multi-threaded execution, the combination of the overall simulation leads to the combined performance improvement of up to 2× (Figure 3c). These results show how the phase I optimizations to Hypre attribute to overall speedups of up to 2×.

3. CPU Performance Analysis and Optimizations: Phase II

This section provides the performance analysis of Hypre after the first phase of optimizations. The second phase focuses on the *communication-centered optimizations* to improve *scalability* at large node counts based on the lessons learned from Phase I.

3.1. Performance Analysis of Phase II

The Theta results show significant improvements in the performance of Hypre and the entire application. However, they also reveal new bottlenecks in Hypre-EP and present opportunities for targeted optimizations to improve performance. Specifically, Hypre-EP stops strong scaling after 64 nodes while Hypre-MPI Only continues to scale (see Figure 3b). The primary reason for the scaling breakdown is the dominating communication cost, as seen from Table 4. Half the solve time is spent waiting for the communication to complete on 64 and 512 nodes. Global locks within the MPI libraries makes `MPI_THREAD_MULTIPLE` communication a major bottleneck. To avoid thread contention on locks in the MPI library, an extra thread is spawned per rank and all communications are funneled through the communication thread. This method works faster than `MPI_THREAD_MULTIPLE` communication, but it serializes all the message exchange and hampers overall performance. The root of this problem can be addressed by an MPI library that does efficient `MPI_THREAD_MULTIPLE` communication and supports MPI Endpoints functionality. Zambre et al. [2] demonstrate the benefits of utilizing network-level parallelism for MPI+threads applications by extending the MPICH implementation to use fine-grained critical sections and virtual communication interfaces (VCIs). VCIs map to the underlying network hardware contexts and

represent dedicated communication channels that threads can map to using MPI Endpoints or existing MPI objects such as communicators and tags. Using VCIs can help exploit underlying network level parallelism and can potentially address the serialization problem observed on Theta.

3.2. Optimizations in Hypre: Phase II

Studies show poor network bandwidth utilization when using fewer ranks per node [17, 18]. Therefore, an application needs to spawn multiple MPI ranks per node to best utilize the network resources. Mapping multiple threads to multiple VCIs within a rank can enable running a single rank per node and still efficiently use network resources. Such a configuration opens up new opportunities for performance optimizations as detailed below.

(a) Non-blocking inter-thread communication. In the funneled communication approach of Phase I, all send/receive requests for EPs, which does not belong to the same rank, are pushed into an inter-process communication queue. The communication queue is monitored by the dedicated communication thread. The communication thread carries on subsequent communication in the background. After pushing the external messages to the communication queue, each EP handles inter-thread communication (i.e., communication with the other EPs of the same rank). Each EP first pushes all the send messages to its *inter-thread send queue* along with copies of the send buffers. Creating a copy allows the EP to continue processing as soon as all of its messages are received without waiting for the send to complete. After queuing all the sends, EPs busy-wait until all receive messages are placed on the queue by respective source EPs. However, if the VCIs enable running one rank per KNL node with 64 threads, it is crucial to have efficient inter-thread communication.

A new non-blocking logic, similar to MPI Isend and Irecv, is introduced to enable efficient inter-thread communication. Each EP simply copies the pointers of send-receive, source/destination thread ids, message size, and tag to a global data structure and returns control back to the calling function. Every send / recv call also tries to progress the communication and copies data from the sender buffer to the receiver buffer if both the values are available. Finally, when `MPI_waitall` is called, EP

keeps checking until all inter-threads send-recv calls are completed. The global data structure allocates different slots to each EP. Every EP writes in its slot only and reads from other threads' slots while copying data. The read-write strategy allows the *lock-free* implementation of inter-thread communication.

Table 5: Wait time in seconds for blocking and non-blocking inter-thread communication
MPI: MPI Only, B: Blocking, NB: Non-blocking

Patch size	$\mathbf{16^3}$			$\mathbf{32^3}$			$\mathbf{64^3}$		
Threads	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**
16	0.042	0.066	0.01	0.063	0.09	0.01	0.16	0.18	0.05
32	0.055	0.2	0.017	0.08	0.3	0.02	0.16	1	0.047
64	0.062	0.49	0.025	0.09	0.56	0.028	0.29	0.73	0.13

As this optimization aims to improve multi-threading, the experiments are conducted on a single rank without MPI in the mix. Hierarchical parallelism is not used to avoid synchronization overheads within each team. Each EP thus is a single thread and not a team of threads. Strong scaling is carried out on a single KNL node for 16, 32, and 64 threads with patch sizes of 16^3, 32^3, and 64^3. The number of patches is set to 64 - one per core. Table 5 compares the communication times (in seconds) of non-blocking inter-thread communication with blocking inter-thread communication from Phase I. The MPI Only implementation is the baseline which spawns 16, 32, and 64 ranks respectively. The measured communication time always increases as the number of threads increases. Non-blocking communication performs 20$\times$ times faster than the blocking communication for 16^3 and 32^3 patches when all 64 threads are spawned. The speedup reduces to 5.6$\times$ for the 64^3 patch since the memory copy time becomes significant due to larger data exchanges. The improved communication impact on the overall solve time is shown in Table 6. The non-blocking code executed 6$\times$, 3$\times$, and 1.3$\times$ faster than the blocking version for 16^3, 32^3, and 64^3 patches respectively on 64 threads. As the patch size increases, computation becomes more dominant and results in lesser speedups. The non-blocking version gave speedups of 1.5$\times$, 1.6$\times$, and 1.5$\times$ over MPI Only code. The speedup over the MPI Only model is because of both vec-

torization and reduction in communication wait time.

Table 6: Solve time in seconds for blocking and non-blocking inter-thread communication
MPI: MPI Only, B: Blocking, NB: Non-blocking

Patch size	$\mathbf{16^3}$			$\mathbf{32^3}$			$\mathbf{64^3}$		
Threads	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**	**MPI**	**B**	**NB**
16	0.32	0.47	0.23	1.6	1.8	0.93	10	10	5.3
32	0.22	0.36	0.15	0.87	1.2	0.52	5.2	6.3	2.8
64	0.15	0.62	0.1	0.5	0.95	0.31	2.8	2.4	1.8

(b) Communication-reducing patch assignment. The assignment of a single multi-threaded rank per node and non-blocking inter-thread communication can reduce MPI communication. MPI communication can be reduced further by a *communication-aware* distribution of patches to ranks. The patch-assignment strategy of Phase I illustrated in Figure 4 divides the total number of patches by the number of ranks and assigns equal-sized chunks of patches sequentially to ranks. This strategy is similar to the storage of 3D arrays in sequential memory. In Figure 4, a grid containing $8\times8\times8$ patches (i.e., a total of 512 patches) is divided among eight ranks. Hypre is run with this configuration on eight KNL nodes with one patch per core. As per the sequential assignment policy, the rank 0 gets the first 64 patches, rank 1 gets the next 64 patches, and so on. Each rank gets one 8×8 *slab* of patches. Assuming 64 EPs per rank, each EP is assigned one patch. Thus, to gather the halo region for the patch, each EP has to communicate with 26 neighboring patches (ignoring face/corner cases) in a three-dimensional grid. Now consider any *internal* patch, which is not on the domain's face. The eight patches surrounding the patch in the same slab belong to the same rank (marked by the same color) and do not require any MPI communication. However, nine front patches and nine rear patches are assigned to different ranks (indicated by a different color), and gathering the halo region involves 18×2 (one send and one receive) = 36 MPI messages. The total number of MPI messages per rank becomes 64 (EPs) $\times$ 36 (messages per EP) = 2304. MPI messages from different EPs of the same rank can not be combined, as this will create a local barrier among threads and thus

hamper the performance. To avoid such a barrier (and to utilize the network to its full capacity), each thread needs to send its MPI messages independently.

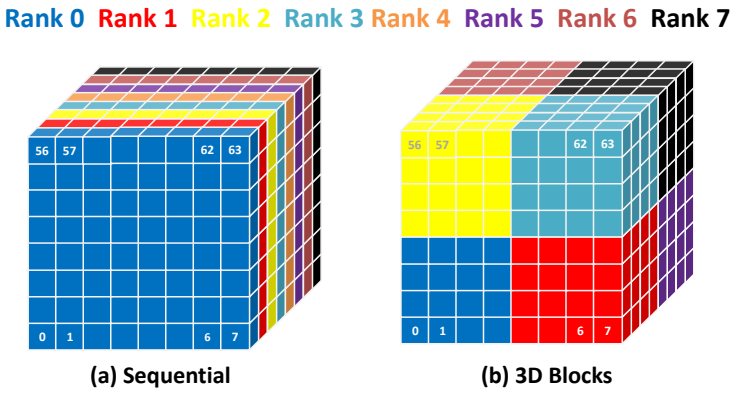


Figure 4: Patch assignment strategies

The number of MPI messages can be reduced by a communication-aware strategy where patches are divided into three-dimensional blocks and assigned to ranks. Such an assignment reduces the number of patches facing those in another rank, and MPI communication is replaced with local inter-thread data exchange. Figure 4 (b) shows the reassignment using 3D blocking. Instead of dividing 512 patches into eight 8×8 slabs, patches are grouped as eight $4\times4\times4$ blocks, and each rank is assigned one block. Thus each EP gets one patch as before, but now each rank has only $16\times3 = 48$ patches facing other ranks. The duplicate patches along the edges are not eliminated because the communication direction is different for different faces. Each patch along the block's face only requires 9×2 MPI messages (one send, one receive). All other communication happens among local threads. Thus the total number of MPI message per rank is $48\times18 = 864$, which results in a $2.6\times$ reduction in the number of MPI messages.

It is straightforward to compute these estimates for the full scale of the Theta supercomputer. Assuming one patch per core, the total number of patches required to run on 4096 nodes is $64\times64\times64$. The sequential patch assignment strategy assigns a *strip* of 64 consecutive patches to each rank. Ignoring face and corner cases of the domain,

each rank generates 24 (only two internal messages) $\times$ 64 (patches facing other ranks) $\times$ 2 (send and receive) = 3072 MPI messages per rank. On the other hand, the 3D blocking patch assignment generates only 9 (other messages will be internal) $\times$ 16 $\times$ 6 (patches facing other ranks) $\times$ 2 (send and receive) = 1728 MPI messages per rank. Thus, the 3D block assignment policy can reduce the number of messages by 1.7$\times$.

The new patch assignment with the threaded implementation replaces MPI messages with more efficient inter-thread data exchange. The new policy can be used for the MPI Only version to replace inter-node MPI messages with intra-node shared-memory MPI messages, but it will not reduce the total number of MPI messages. Because inter-thread communication is more efficient than shared-memory MPI communication, the new patch assignment is expected to benefit the EP version more than the MPI Only version.

(c) Exposing logical MPI communication parallelism. MPI libraries have recently made significant strides towards improving the communication performance of MPI_THREAD_MULTIPLE to match that of MPI everywhere [19, 20, 2]. A key contributing factor to this improved performance has been the mapping of independent MPI communication to network-level parallelism available on modern interconnects. These new libraries, however, are helpless if applications do not distinguish between operations that are ordered and those that are independent. Hence, to leverage the high-speed multithreaded communication in these new libraries for Hypre-EP, it is imperative to expose the communication independence through MPI.

In Hypre-EP, no ordering constraints apply for operations originating from different Endpoints, that is, each Endpoint's communication is *logically parallel*. A straightforward way to expose this logical communication parallelism with the MPI Endpoints proposal is to use a distinct MPI Endpoint for each Hypre EP. The MPI forum, however, has suspended the MPI Endpoints proposal since existing MPI mechanisms, such as communicators, tags, and windows, can expose the same amount of parallelism as MPI Endpoints. In fact, Hypre-EP's mechanism of encoding the sender and receiver thread IDs into the MPI tag to distinguish between messages targeting the same MPI rank can double as logical communication parallelism information. But, using tags is not sufficient with the existing MPI-3.1 standard because of the possibility of wildcards

on the receive operations even though Hypre-EP does not use wildcards. The upcoming MPI-4.0 standard, however, introduces new Info hints that allow applications, like Hypre-EP, to inform the MPI library that it does not use wildcards.

Hypre-EP leveraged the new hints in the draft MPI-4.0 standard (mpi_assert_no_any_source and mpi_assert_no_any_tag) which allowed the MPICH implementation used in this work to utilize the encoded parallelism information in the tags to map to its multi-VCI infrastructure. Like MPI Endpoints, tags with hints expose all of the available communication parallelism information. In the near future, most MPI libraries will be capable of mapping logical communication parallelism to the underlying network parallelism be it through existing MPI objects or through MPI Endpoints.

(d) One rank per node. As discussed earlier, VCIs allows efficient use of the network resources even with a single multi-threaded rank per node. As a result, intra-node MPI communication can be replaced with more efficient inter-thread communication and can result in a faster runtime. A simple change in the runtime configuration can allow running one rank per node with all the cores on the node utilized by the rank. The configuration is not an enhancement by itself, but it maximizes the impact from the previous optimizations such as inter-thread communication and communication-reducing patch assignment.

The second impact of using more threads is in the RMCRT component within Uintah. As seen from Phase I results, multi-threaded RMCRT with 16 threads per rank performs $2\times$ to $4\times$ faster than the MPI Only version. The reduction in the number of ranks reduces the cost of the all-to-all MPI communication in RMCRT. Multi-threading can thus provide a higher payoff in RMCRT if the number of threads per rank is set to 64 instead of 16.

(e) Hierarchical parallelism. The final optimization is to enable hierarchical parallelism. As described in Section 2.3c, the combination of EP and hierarchical parallelism can minimize both MPI wait time and the thread synchronization overhead. The experiments show that the patch size and the team size are directly proportional, i.e., a smaller team size performs better for a small patch size. As the patch size grows, the workload of a `parallel_for` increases, and the additional synchronization cost due

to a larger team size can be justified by improved overall performance. For a 64^3 patch, a team size of 4 results in the best performance.

3.3. Experimental Setup on Bebop

The Phase I experiments proved the effectiveness of multi-threaded execution on the overall simulation by speeding up both RMCRT and Hypre. As the focus of this work is Hypre, the same experiments with RMCRT are not repeated. Instead, a standalone mini-app is built, which replicates Uintah's calls to Hypre and produces the same output. The mini-app provides greater flexibility for in-depth analysis without incorporating other Uintah complexities such as task graphs, task schedulers, etc. All the experiments in Phase II are run using the mini-app.

The multi-VCI effort in MPICH currently only supports the Intel Omni-Path and Mellanox InfiniBand interconnects. Multi-VCI support for Theta's Aries interconnect is in progress. Hence, a new set of experiments is run on Argonne National Laboratory's Bebop cluster which uses the Intel Omni-Path interconnect. Bebop features 352 Intel Xeon Phi 7230 (KNL) nodes with 64 cores and 16 GB MCDRAM per node. These nodes support AVX512 vector instructions. MCDRAM is set in the cache-quadrant mode for all the experiments. A customized version of MPICH with fine-grained lock and VCI support is used to compile the code along with Intel Parallel Studio 18.0.5. The flags provided for Hypre configuration include: "-std=c++11 -fp-model precise -g -O2 -xMIC-AVX512 -fPIC -fopenmp".

Multiple strong scaling runs are carried out to evaluate the performance of the new communication-centered optimizations for a small problem. Funneled communication from the Phase I experiments is now disabled because VCIs can efficiently perform multi-threaded MPI communication. Hierarchical parallelism from Phase I is also disabled initially to correctly measure the impact of using VCIs. Thus the new baseline is a simple endpoint version with vectorization. Without funneled communication and hierarchical parallelism, the new baseline is expected to perform slower than both the MPI Only version and the most optimized version from Phase I. However, the use of this baseline allows the measurement of the impact of each optimization incrementally. With this aim in mind, the experiments are carried out using the following levels of

optimizations from Section 3.2:

- EP Baseline: Basic MPI End Points without funneled communication and hierarchical parallelism.
- O1: Baseline + Non-blocking inter-thread communication.
- O2: O1 + Communication-reducing patch assignment.
- O3: O2 + VCIs
- O4: O3 + VCIs + one rank per node with 64 threads per rank.
- O5: O4 + Hierarchical parallelism.

The baseline, and the O1, O2, and O3 optimizations are run with four ranks per node, and 16 EPs per rank. The O4 and O5 optimizations are run using one rank per node with 64 EPs per rank. The O5 enhancement uses different combinations of the number of EPs (i.e., thread teams) $\times$ the number of threads per team (i.e., team size). The best performing results are chosen from the various combinations: 64×1, 32×2, and 16×4. The MPI Only version is run with 64 ranks per node (i.e., one rank per core) as a reference.

The number of patches is chosen such that there will be one patch per core at the end of strong scaling. Strong scaling is carried out at three different levels:

1. A **small** problem of size 512^3 divided among 512 patches ($8\times8\times8$) is run on 2, 4, and 8 nodes to incrementally test the various optimizations.
2. A **medium** problem consists 1024^3 cells divided among 4096 patches ($16\times16\times16$) and is executed on 16, 32, and 64 nodes.
3. A **large** problem consists of 2048^3 cells decomposed into 32k patches ($32\times32\times32$) and is run on 128 and 256 nodes of Bebop, respectively. Unfortunately, the Bebop cluster does not have 512 nodes and the last step in strong scaling cannot be performed.

In all the cases, the patch size remains fixed, i.e., 64^3. Medium and large-scale problems are run on a larger number of nodes to compare the most optimized EP version with the MPI Only model. Each problem is run for ten timesteps and for 60 CG-solver iterations per timestep. The setup time is discarded as it is done only during the zeroth timestep. Average solve time and communication time from the rest of the timesteps are measured and presented in the subsequent sections.

3.4. Results and Evaluation on Bebop

Table 7: Solve and communication times in seconds for the *small* problem with different optimizations.

	Solve time							**Communication time**						
Nodes	**MPI**	**EP**	**O1**	**O2**	**O3**	**O4**	**O5**	**MPI**	**EP**	**O1**	**O2**	**O3**	**O4**	**O5**
2	39	39	33	29	36	24	24	6.5	20	14	9.2	16	2.4	1.6
4	24	32	21	20	34	14	12	7.4	23	12	10	24	3	1.2
8	17	28	20	16	28	9.7	7	7.1	24	15	12	22	3.5	1.4

MPI: MPI Only, 64 ranks/node; EP: EP baseline, no funneled comm, 4 ranks/node, 16 EPs/rank; O1: EP + Non-blocking inter-thread comm; O2: O1 + Comm reducing patch assignment; O3: O2 + VCIs; O4: O3 + 1 rank/node, 64 EPs/rank; O5: O4 + Hierarchical Parallelism: 1 rank/node, 16 EPs per rank, 4 worker threads per EP

Table 7 compares the strong scaling of solve and communication times of the different optimizations for a 512^3 problem. The MPI Only version provides a reference. As expected, turning off funneled communication and hierarchical parallelism slowed down the Hypre-EP baseline more than Hypre-MPI Only. However, overall performance improved with each optimization and the final version is faster than the MPI Only version. Enabling non-blocking inter-thread communication improved EP-baseline performance by 1.3–1.4×, and EP now runs within +/- 15% of the MPI Only version. The second optimization–communication-reducing patch assignment–did not show much impact on two and four nodes but improves performance by 20% on eight nodes, which makes Hypre-EP perform as fast or even faster than Hypre-MPI Only. Introducing VCIs, however, proved to be inefficient. The VCI abstractions in the MPICH

library are not yet tuned for intra-node shared memory communications, which resulted in the performance drop shown in the O3 column of Table 7. The situation is remedied by using a single rank per node, as indicated by column O4, which is the primary goal in introducing VCIs, and the speedup over the MPI Only version improves by 1.7× on four and eight nodes. Hierarchical parallelism provides an additional performance increase as indicated by the column O5 and the final improvement over the MPI Only version reaches 2× and 2.4× on four and eight nodes, respectively. The Phase I optimizations results in 1.3–1.4× speedup for the same setup, which indicates that the Phase II optimizations further improve performance by 1.7× over Phase I. The main reason behind this significant improvement is the reduced communication wait time. The O5 optimization in Table 7 shows up to a 5× reduction in communication wait time than the MPI Only version. Figure 5 shows strong scaling of Hypre-MPI Only

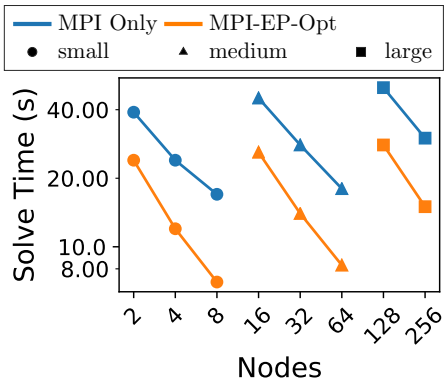


Figure 5: Strong Scaling of Solve time

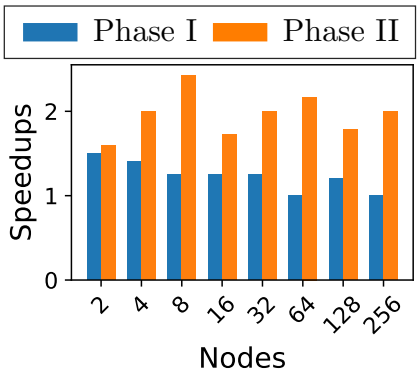


Figure 6: Speedups over Hypre-MPI Only

and Hypre-EP-O5, the most optimized EP version, up to 64 nodes. The performance improvement for a 512^3 problem on two, four, and eight nodes are 1.6×, 2×, and 2.4×, respectively. The 1024^3 problem shows improvements of 1.7×, 2×, and 2.1× on 16, 32, and 64 nodes, respectively. The large problem of size 2048^3 run on 128 and 256 nodes of Bebop demonstrated improvements of 1.78× and 2×, respectively. The Phase I experiments are run on Theta and Phase II experiments on the Bebop cluster. Although both the machines deploy Intel's KNL processors, the underlying networks are different. Hence, the results can not be directly compared with each other. There-

fore, the performance of Hypre-EP over Hypre-MPI Only is compared between Phase I and Phase II, as shown in Figure 6. Recall that Phase I hits the strong scaling limit at 64 nodes and 256 nodes for the medium and large size problems, respectively, and the results show no improvements over the MPI Only version. Furthermore, the speedups during Phase I decreased for the given problem size as strong scaling progressed. Phase II optimizations address this communication bottleneck and result in faster overall execution time and results into speedups up to 2× on 256 nodes. Table 8 compares the

Table 8: Percent communication wait time for MPI EP in Phase I vs. Phase II.

Nodes	2	4	8	16	32	64	128	256
Phase I	10%	11%	21%	19%	24%	50%	30%	42%
Phase II	7%	10%	20%	12%	19%	28%	18%	27%

communication wait times of the experiments in Phase I and Phase II. Phase II shows a significant reduction in the percentage of time spent in communication wait routines on 16 through 256 nodes. As a result, the speedups over the MPI Only version increases for a given problem size as the number of nodes increases, which indicates improved strong scaling.

4. GPU Performance Characterization and Enhancements

While Hypre has had CUDA support from version 2.13.0, version 2.15.0 is used here to characterize performance, to profile for bottlenecks and to optimize the solver code. The GPU experiments are carried out on LLNL's Lassen cluster. Each node is equipped with two IBM Power9 CPUs with 22 cores each and four Nvidia V100 GPUs. Hypre and Uintah both were compiled using gcc 4.9.3 and cuda 10.1.243.

4.1. Performance Characterization and Optimizations for GPUs

The initial performance characterization was done on 16 GPUs of Lassen using a standalone mini-app which called Hypre to solve a simple Laplace equation and run for 20 iterations. GPU strong scaling is carried out using 16 "super-patches" of varying sizes $44^3, 64^3$ and 128^3. The observed GPU performance is evaluated against the

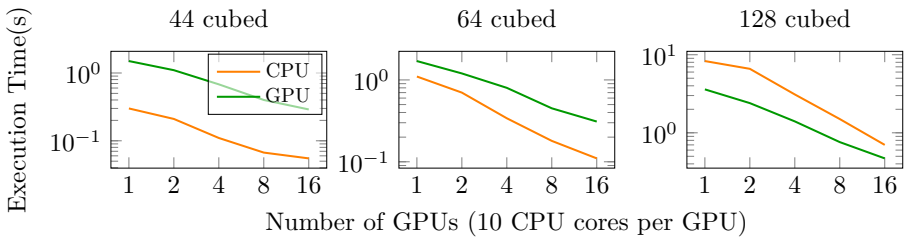


Figure 7: GPU performance variation based on patch size

Table 9: Top five longest running kernels before and after merging

Before merging			**After Merging**		
Name	**Calls**	**Avg Time**	**Name**	**Calls**	**Avg Time**
MatVec	3808	29.067us	MatVec	3808	29.040us
Relax1	2464	22.453us	Relax1	2464	22.463us
Relax0	2352	19.197us	Relax0	2352	19.126us
InitComm	20656	1.8650us	Axpy	1660	21.484us
FinComm	20688	1.8310us	Memcpy-HtoD	12862	2.0750us

corresponding CPU performance, which is obtained using the MPI only CPU version of Hypre. Thus, corresponding to every GPU, 10 CPU ranks are spawned and super-patches are decomposed smaller patches into smaller patches to feed each rank, keeping the total amount of work the same. Figure 7 shows the CPU performs 5x faster than the GPU for patch size 44^3. Although 64^3 patches decrease the gap, it takes the patch size of 128^3 for GPU to justify overheads of data transfers and launch overheads and deliver better performance than CPU. Based on this observation, all further work as carried out using 128^3 patches. HPCToolkit and Nvidia nvprof were used to profile CPU and GPU executions. The sum of all GPU kernel execution time shown by nvprof was around 500ms, while the total execution time was 1.6 seconds. Thus the real computation work was only 30% and nearly 70% of the time was spent in the bottlenecks other than GPU kernels. Hence, tuning individual kernels would not help as much. This prompted the need for CPU profiling which revealed about 30 to 40% time consumed in for MPI wait for sparse matrix-vector multiplication and relaxation

routines. Another 30 to 40% of solve time was spent in the cuda kernel launch overhead. It should be noted that although the GPU kernels are executed asynchronously, the launching itself is synchronous. Thus to justify the launching overhead, the kernel execution time should be at least 10μs - the launch overhead of the kernel on V100 (which was shown in the nvprof output).

Table 9 shows the top five longest running kernels for the solve time of 128^3 patches on 16 GPUs with one patch per GPU. InitComm and FinComm kernels which are used to pack and unpack MPI buffers are fourth and fifth in the list. The combined timing of these two kernels can take them to the second position. More interestingly, together these kernels are called for 41,344 times, but the average execution time per kernel execution is just 1.8 μs. On the other hand the launch overhead of the kernel on V100 is 10μs (which was revealed in the profile output). Thus the launch overhead of pack-unpack kernels consumes 0.4 seconds of 1.6 seconds (25%) of total execution time.

The existing implementation iterates over neighboring dependencies and patches and launches the kernel to copy required cells from the patch into the MPI buffer (or vice a versa). This results in thousands of kernel launches as shown in Table 9, but the work per launch remains minimal due to a simple copying of few cells. The problem can be fixed by fusing such kernel launches - at least for a single communication instance. To remedy the situation, the CPU code first iterates over all the dependencies to be processed and creates a buffer of source and destination pointers along with indexing information. At the end, all the buffers are copied into GPU's constant memory cache and the pack (or unpack) cuda kernel is launched *only once* instead of launching it for every dependency. After the fix InitComm and FinComm disappeared from the top five longest running kernels as shown in Table 9. The combined number of calls for InitComm and FinComm reduced from 41,344 to 8338. As a result, the communication routines perform 3x faster than before and the overall speedup in solve time achieved was around 20%. The modified code adds some overhead due to copying value to the GPU constant memory, which is reflected Memcpy-HtoD being called 12862 times compared to 4524 times earlier, but still the new code performs faster.

With the first major bottleneck resolved, the second round of profiling using HPC-Toolkit showed that the MPI wait time for matrix vector multiplication and for relax-

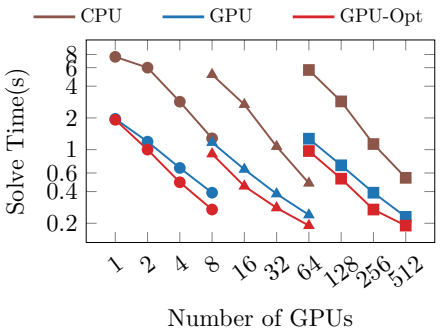


Figure 8: Strong Scaling of Solve time

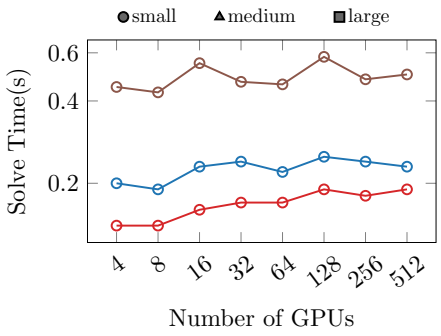


Figure 9: Weak Scaling of Solve time

ation routines was now more than 60%. The problem is partially overcome by using cuda aware MPI supported on Lassen. The updated code directly passes GPU pointers to the MPI routines and avoids copying data between host and device. This decreased the communication wait time to 40 to 50% and resulted in an extra speedup of 10%.

4.2. GPU Experiments on LLNL's Lassen

The GPU experiments were carried out on LLNL's Lassen cluster. Each node is equipped with two IBM Power9 CPUs with 22 cores each and four Nvidia V100 GPUs. Hypre and Uintah both were compiled using gcc 4.9.3 and cuda 10.1.243 with compiler flags "-fPIC -O2 -g -std=c++11 –expt-extended-lambda".

Strong and weak scaling experiments on Lassen were run by calling Hypre from Uintah (instead of mini-app) and the real equations originating from combustion simulations were passed to generate the solve for the pressure at each mesh cell. Strong scaling experiments were conducted using three different mesh sizes: small (512x256x256), medium (512^3) and large (1024^3). Each mesh is divided among patches of size 128^3 - such a way that each GPU gets one patch at the end of the strong scaling. CPU scaling was carried out by assigning one MPI rank to the every available CPU core (40 CPU cores/node) and by decomposing the mesh into smaller patches to feed each rank.

4.3. GPU Results on Lassen

The strong scaling plot in Figure 8 shows GPU version performs 4x faster than CPU version in the initial stage of strong scaling when the compute workload per GPU

is more. As the GPU version performs better than the CPU version, it runs out of compute work sooner than the CPU version and the scaling breaks down with speedup reduced to 2.3x. Similarly, the optimized GPU version performs up to 6x faster than the CPU version (or 1.44x faster than the baseline GPU version) with the heavy workload. As the strong scaling progresses, the speedup by the optimized version against CPU reduces to 3x (or 1.26x against baseline GPU version). The communication wait time of both GPU versions is reduced by 4x to 5x as the number of ranks is reduced by ten times (not shown for brevity). Thanks to faster computations, the optimized GPU version spends 15 to 25% more time in waiting for MPI compared to the baseline GPU version.

The weak scaling was carried out using one 128^3 patch per GPU (or distributed among ten CPU cores) from four GPUs to 512 GPUs. Figure 9 shows good weak scaling for all three versions. The GPU version shows 2.2x to 2.8x speedup and the optimized GPU code performs 2.6x to 3.4x better than the CPU version.

Preliminary experiments with the MPI EP model on Lassen showed that the MPI EP CPU version performed as well as the MPI Only CPU version (not shown in Figure 8 for brevity). Work is in progress to improve GPU utilization by introducing the MPI EP model for the GPU version and assigning different CUDA streams to different endpoints which may improve overall performance.

5. Conclusions and Future Work

This paper shows that the MPI-Endpoint approach makes a threaded version of Hypre up to 2.4x faster than the MPI-only version. One of the bottlenecks for the MPI EP version was locks within MPI. This problem was resolved using an MPICH implementation that supports fined-grained locks and allows threads to exploit network level parallelism using VCIs. It proves the impact of using VCIs on a real-life application and shows how a combination of a well designed multi-threaded implementation supported by VCIs can lead to better performance on modern architectures. The overall performance improvement of 1.7x–2.4x and a communication wait time reduction up to 5x can prove to be a significant advantage on CPU based supercomputers such as

Fugaku, and TACC Frontera in the future. Other multi-threaded applications can benefit from the combination of VCIs supported EndPoints and hierarchical parallelism to achieve overall speedup as demonstrated by Uintah and Hypre on Theta and Bebop.

Similarly, improved GPU speedups can help in gaining overall speedups for other Hypre-cuda users.

Not only Hypre, but Uintah may also also benefit from VCIs, and it will be exciting to see if VCIs can be used to avoid any communication bottlenecks in Uintah. On GPUs the current optimized version shows around 40 to 50% time is consumed in waiting for MPI communication during sparse matrix vector multiplication and relaxation routines. If the computations and communications are overlapped, then a new kernel needs to be launched to undertake the dependent computations after the communication is completed. As these kernels do not have enough work to justify the launch, this resulted into slightly slower overall execution times during the initial experiments of overlapping communications. Similar behavior was observed by [21]. A possible solution is to collect kernels as "functors" and to launch a single kernel later, which calls these functors one after another as a function call. Another option for speeding up the algorithm is to use communication avoiding approaches e.g., see [22] which uses a multi-grid preconditioner and spends less than 10% of the solve time in the global MPI reductions on Summit. As this work here also used a multi-grid preconditioner [23], similar behavior was observed in our experiments and the global reduction in the CG algorithm is not a major bottleneck so far. However, these options will be revisited when applying the code to full scale combustion problems at Exascale.

References

[1] D. Sahasrabudhe, M. Berzins, Improving Performance of the Hypre Iterative Solver for Uintah Combustion Codes on Manycore Architectures Using MPI Endpoints and Kernel Consolidation, in: Computational Science – ICCS 2020, Springer International Publishing, Cham, 2020, pp. 175–190.

[2] R. Zambre, A. Chandramowliswharan, P. Balaji, How I Learned to Stop Worrying about User-Visible Endpoints and Love MPI, in: Proceedings of the 34th ACM

International Conference on Supercomputing, ICS '20, Association for Computing Machinery, New York, NY, USA, 2020.

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

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

[5] R. D. Falgout, J. E. Jones, U. M. Yang, Pursuing scalability for hypre's conceptual interfaces, ACM Transactions on Mathematical Software (TOMS) 31 (3) (2005) 326–350.

[6] A. H. Baker, R. D. Falgout, T. V. Kolev, U. M. Yang, Scaling hypre's multigrid solvers to 100,000 cores, in: High-Performance Scientific Computing, Springer, 2012, pp. 261–279.

[7] 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, 2018, pp. 219–240.

[8] J. K. Holmen, B. Peterson, M. Berzins, An approach for indirectly adopting a performance portability layer in large legacy codes, in: 2nd International Workshop on Performance, Portability, and Productivity in HPC (P3HPC), In conjunction with SC19, 2019.

[9] A. Humphrey, M. Berzins, An evaluation of an asynchronous task based dataflow approach for uintah, in: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), Vol. 2, 2019, pp. 652–657.

[10] A. Baker, R. Falgout, T. Kolev, U. Yang, Scaling hypre's multigrid solvers to 100,000 cores, in: M. Berry, K. Gallivan, E. Gallopoulos, A. Gram, B. Philippe, Y. Saad, F. Saied (Eds.), High-Performance Scientific Computing, Springer London, 2012, pp. 261–279.

[11] H. Gahvari, W. Gropp, K. E. Jordan, M. Schulz, U. M. Yang, Modeling the Performance of an Algebraic Multigrid Cycle Using Hybrid MPI/OpenMP, in: 2012 41st International Conference on Parallel Processing, 2012, pp. 128–137.

[12] J. Dinan, R. E. Grant, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling communication concurrency through flexible MPI endpoints, The International Journal of High Performance Computing Applications 28 (4) (2014) 390–405.

[13] R. Zambre, A. Chandramowlishwaran, P. Balaji, Scalable Communication Endpoints for MPI+Threads Applications, in: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), 2018, pp. 803–812.

[14] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, R. Thakur, Enabling MPI interoperability through flexible communication endpoints, in: Proceedings of the 20th European MPI Users' Group Meeting, 2013, pp. 13–18.

[15] A. Humphrey, T. Harman, M. Berzins, P. Smith, A scalable algorithm for radiative heat transfer using reverse monte carlo ray tracing, in: J. M. Kunkel, T. Ludwig (Eds.), High Performance Computing, Vol. 9137 of Lecture Notes in Computer Science, Springer International Publishing, 2015, pp. 212–230.

[16] B. Peterson, A. Humphrey, J. H. T. Harman, M. Berzins, D. Sunderland, H. Edwards, Demonstrating GPU Code Portability and Scalability for Radiative Heat Transfer Computations, Journal of Computational Science.

[17] Argonne Leadership Computing Facility. 2017. Theta Web Page. (2017)., https://www.alcf.anl.gov/support-center/theta/aries-network-theta.

[18] B. Alverson, E. Froese, L. Kaplan, D. Roweth, Cray XC series network, Cray Inc., White Paper WP-Aries01-1112.

[19] T. Patinyasakdikul, D. Eberius, G. Bosilca, N. Hjelm, Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs, in: 2019 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2019.

[20] Intel® MPI Multiple Endpoints Support, https://software.intel.com/en-us/mpi-developer-guide-linux-multiple-endpoints-support.

[21] Y. Ali, N. Onodera, Y. Idomura, T. Ina, GPU Acceleration of Communication Avoiding Chebyshev Basis Conjugate Gradient Solver for Multiphase CFD Simulations, in: 2019 IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (ScalA), IEEE, 2019, pp. 1–8.

[22] Y. Idomura, T. Ina, S. Yamashita, N. Onodera, S. Yamada, T. Imamura, Communication avoiding multigrid preconditioned conjugate gradient method for extreme scale multiphase CFD simulations, in: 2018 IEEE/ACM 9th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems (scalA), IEEE, 2018, pp. 17–24.

[23] J. Schmidt, M. Berzins, J. Thornock, T. Saad, J. Sutherland, Large Scale Parallel Solution of Incompressible Flow Problems using Uintah and hypre, SCI Technical Report UUSCI-2012-002, SCI Institute, University of Utah (2012).