Improving Performance of the Hypre Iterative Solver for Uintah Combustion Codes on Manycore Architectures Using MPI Endpoints and Kernel Consolidation

Damodar Sahasrabudhe(✉) and Martin Berzins

SCI Institute, University of Utah, Salt Lake City, UT, USA
{damodars,mb}@sci.utah.edu

Abstract. The solution of large-scale combustion problems with codes such as the Arches component of Uintah on next generation computer architectures requires the use of a many and multi-core threaded approach and/or GPUs to achieve performance. Such codes often use a low-Mach number approximation, that require the iterative solution of a large system of linear equations at every time step. While the discretization routines in such a code can be improved by the use of, say, OpenMP or Cuda Approaches, it is important that the linear solver be able to perform well too. For Uintah the Hypre iterative solver has proved to solve such systems in a scalable way. The use of Hypre with OpenMP leads to at least 2x *slowdowns* due to OpenMP overheads, however. This behavior is analyzed and a solution proposed by using the MPI Endpoints approach is implemented within Hypre, where each team of threads acts as a different MPI rank. This approach minimized OpenMP synchronization overhead, avoided slowdowns, performed as fast or (up to 1.5x) faster than Hypre's MPI only version, and allowed the rest of Uintah to be optimized using OpenMP. Profiling of the GPU version of Hypre showed 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. The overall speedup of 1.26x to 1.44x was observed compared to the baseline GPU implementation.

Keywords: Hypre · OpenMP · GPUs · MPI End Point

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.

V. V. Krzhizhanovskaya et al. (Eds.): ICCS 2020, LNCS 12137, pp. 175–190, 2020.
https://doi.org/10.1007/978-3-030-50371-0_13

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 512 K cores on the NSF Stampede, DOE Titan, and DOE Mira 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 is used to simulate heat, mass, and momentum transport in reacting flows by using a low Mach number ($Ma < 0.3$) variable density formulation [14]. 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 [14]. Hypre supports different iterative and multigrid methods, has a long history of scaling well [2,5] and has successfully weak scaled up to 500000 cores when used with Uintah [11].

While Uintah simulations were carried out [3] on DOE Mira and Titan systems [11], the next generation of simulations will be run on many core architectures such as DOE's Theta, NSFs Frontera, Riken's Fugaku and on GPU architectures such as DOEs Lassen Summit, Frontier 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. For many cores a common approach is to use a combination of MPI and OpenMP to achieve this massive parallelism. In the case of GPUs an offload of the OpenMP parallel region to GPU with CUDA or OpenMP 4.5 may be used. It is also possible to use portability layers such as Kokkos [7] to automate the process of using either OpenMP or Cuda. The MPI-only configuration for Uintah is to have one single threaded rank per core and one patch per rank. In contrast, the Uintah's Unified Task Scheduler was developed to leverage multi-threading and also to support GPUs [8]. Work is in progress to implement portable multi-threaded Kokkos - OpenMP and Kokkos - Cuda [7] 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 with multiple patches, which are distributed among teams. Teams of threads process patches in parallel (task parallelism) while threads within a team work on a single patch (data parallelism). This design has proven useful on many core systems and in conjunction with Kokkos has led to dramatic improvements in performance [7].

The challenge addressed here is to make sure that similar improvements may be seen with Uintah's use of Hypre and its Structured Grid Interface (Struct) at the very least performs as well in a threaded environment as in the MPI case. Hypre's structured multigrid solver, PFMG, [2] 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. [2] report that various version of PFMG are between 2.5 and 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 [14].

The Eq. (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, the large number of variables, for example 6.4 Billion in [14], 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 multi-threaded 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 Moving Hypre to New Architectures

In moving the Hypre to manycore architectures OpenMP was introduced to support multithreading [6]. However, in contrast to the results in [6], when using Uintah with Hypre in the case of one MPI process and OpenMP with multiple cores and mesh patches, a dramatic slowdown of up to 3x to 8x slowdown was experienced when using Hypre with Uintah as in a multi-threaded environment, as compared to the MPI-only version. Similar observations were made by Baker using a test problem with PFMG solver and up to 64 patches per rank and slowdown of 8x to 10x was observed between the MPI-only and MPI+OpenMP versions [2]. The potential challenges with OpenMP and Hypre either force Uintah with Hypre to singlethreaded (MPI only) version or use OpenMP with one patch per rank. This defeats the purpose of using OpenMP.

This work will show that the root cause of the slowdown to be the use of OpenMP pragmas at the innermost level of the loop structure. However the obvious solution of moving these OpenMP pragmas to a higher loop level does not seem to offer the needed performance either. The solution adopted here is to use a variant of an alternate threading model "MPI scalable Endpoints" [4,16] to solve the problem and to achieve a speedup consistent with the observed results

of [2,6]. The approach described here is referred to as "MPI Endpoints", and abbreviated as MPI Ep, requires overriding MPI calls to simulate MPI behavior, parallelizing packing and unpacking of MPI buffers.

In optimizing Hypre performance for GPUs, Hypre 2.15.0 was run as a baseline code on Nvidia V100 GPUs, to characterize 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 was fixed by fusing these micro kernels together and using GPU's constant cache memory. Finally, Hypre was modified to leverage Cuda-aware MPI on Lassen cluster which gives extra 10% boost.

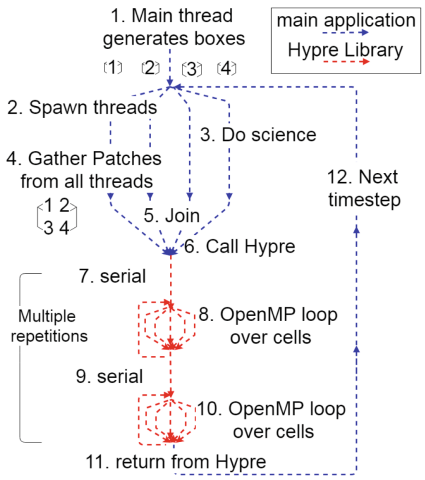
The main contributions of this work are: (i) Introduce MPI EP model in Hypre to avoid slowdowns observed in the OpenMP version, which can achieve faster overall performance in the future while running the full simulation using multi-threaded task scheduler within Uintah AMT. (ii) Optimize the Cuda version of Hypre to improve CPU to GPU speedups ranging from 2.3x to 4x in the baseline version to the range of 3x to 6x in the optimized version, which can benefit the future large-scale combustion simulations on GPU based supercomputers.

2 Analysis of and Remedies for OpenMP Slowdown

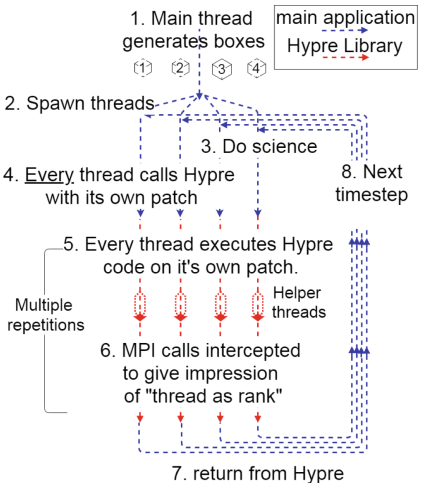
The slowdown of OpenMP was investigated by profiling of Hypre using the PFMG preconditioner and the PCG solver with a representative standalone code that solves a 3D Laplace equation on a regular mesh, using a 27 point stencil. Intel's Vtune amplifier and gprof were used to profile on a single node KNL with 64 cores. The MPI-Only version of the code was executed with 64 single threaded ranks and the MPI + OpenMP version used 1×64, 2×32, 4×16, 8×8 and 16×4 ranks and threads, respectively. The focus was on the solve step that is run at every time step rather than the setup stage that is only called once. This example mimicked the use of Hypre in Uintah in that each MPI rank derived its own patches (Hypre boxes) based on the rank and allocated the required data structures accordingly. Each rank owned from a minimum of 4 patches to a maximum of 128 patches and each patch was then initialized by its respective rank. The Struct interface of Hypre was then called - first to carry on the setup and then to solve the equations. The solve step was repeated up to 10 times to simulate timesteps in Uintah by slightly modifying cell values every time. Then each test problem used different combinations of domain and patch sizes: a 64^3 or 128^3 domain was used with 4^3 patches of sizes 16^3 or 32^3. A 128^3 or 256^3 domain was 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 were tried and compared against the MPI-only version. Each solve step took about 10 iterations to converge on average.

The main performance bottlenecks were noted as follows.

(a) **OpenMP fork-join overhead**. Figure 1a shows the code structure of how an application (Uintah) calls Hypre. Uintah spawns its own threads, generates patches, and executes tasks scheduled on these patches. When Uintah



(a) Existing Control Flow of Hypre



(b) New Control Flow of Hypre

Fig. 1. Software design of hypre

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 shows that Hypre solve took 595 s. Of this time the OpenMP fork-join overhead was 479 s and spin time was 12 s.

The PFMG-CG algorithm calls 1000 s of "micro-kernels" during the solve step. Each micro kernel performs lightweight operations such matrix vector multiplication, scalar multiplication, relaxation, etc. and uses OpenMP to parallelize over the patch cells. However, the light workload does not offset the overhead of the OpenMP thread barrier at the end of every parallel for and results into 6x performance degradation. As a result, Hypre does not benefit from multiple threads and cores, with a performance degradation from OpenMP that grows with the number of: OpenMP threads per rank, patches per rank and points per patch.

(b) **Load imbalance due to serial sections**. Profiling detected three main serial parts - namely: 1. Packing and unpacking of buffers before and after MPI communication, 2. MPI communication and 3. Local data halo exchanges. Furthermore, the main thread has to do these tasks on behalf of worker threads while in the MPI-only version, each rank processes its own data and, of course, it does not have to wait for other threads.

(c) **Failure of auto-vectorization**. Hypre has "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 needed 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.

2.1 Restructuring OpenMP Loops

One obvious solution to the bottlenecks identified above is to place pragmas at the outermost loop possible, namely the loop at "patch" level. This was tested for the Hypre function `hypre_PointRelax`. Table 1 shows timings for the MPI only version, default MPI + OpenMP version with OpenMP pragmas around cell loops and the modified OpenMP version where OpenMP pragmas were moved from cells to mesh patches, thus assigning one or more mesh patches to every thread. The shifting of OpenMP pragma gave a performance boost of 1.75x. However this is still 2x slower than the MPI only version. The final result in Table 1 is for the new approach suggested here that performs as well as MPI and is now described.

Table 1. Comparison of MPI vs OpenMP execution time(s) using 64 32^3 Mesh Patches

Hypre run time configuration	Runtime (s)
MPI Only 64 ranks	1.45
Default 4 ranks, each with 16 threads, OpenMP on cells loop	5.61
Modified 4 ranks, each with 16 threads, OpenMP on boxes loop	3.19
MPI Endpoints: 4 ranks each with 4 teams each with 4 threads	1.56

The MPI Endpoints approach adopted to overcome these challenges is shown in Fig. 1b. In this new approach, each Uintah "team of threads" acts: independently as if it is a separate rank (also known as MPI End Point or EP) and calls Hypre, passing its own patches. Each team processes its own patches and communicates with other real and virtual ranks. The mapping between teams and ranks is *virtual rank = real rank * number of teams + team id*. MPI wrappers are updated to 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 code behaves as if it is MPI only version. The smaller team size (compared to the entire rank) minimizes overhead incurred in fork join in the existing OpenMP implementation, yet can exploit data parallelism.

The design and implementation of this approach posed the following challenges.

(a) **Race Conditions:** All global and static variables were converted to thread_local variables to avoid race conditions.
(b) **MPI Conflicts:** A potentially challenging problem was to avoid MPI conflicts due to threads. In Hypre only the main thread was designed to handle all MPI communications. With the MPI Endpoints approach, each team is required to make its own MPI calls. As Hypre already has MPI wrappers in place for all MPI functions, adding some code in every wrapper function to convert between a virtual rank and a real rank and to synchronize teams during MPI reductions was enough to avoid MPI conflicts.

(c) **Locks within MPI:** The MPICH implementation used as a base for Intel MPI and Cray MPI for the DOE Theta system uses global locks. As a result, only one thread can be inside the MPI library for most of the MPI functions. This is a potential problem for the new approach as the number of threads per rank are increased. To overcome the problem, one extra thread was spawned and all the communication funneled through the communication thread during the solve phase. This method provides a minimum thread wait time and gives the best throughput.

2.2 Optimizations in Hypre

The implementation of this approach needed following changes:

```
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;
}
```

Fig. 2. Pseudo code of MPI EP wrapper for MPI_Comm_rank

(a) **MPI Endpoint:** The approach adopted 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. Also MPI reduce and probe calls need extra processing. These changes are now described below.

(i) **`MPI_Comm_rank:`** this command was mapped by using the formula above relating ranks and teams. Figure 2 shows pseudo code used to convert the real MPI rank to the virtual MPI EP rank using formula "mpi_rank * g_num_teams + tl_team_id". The global variable g_num_teams and the thread local variable tl_team_id are set to the number of teams and the team id during initialization. Thus the each end point gets an impression of a standalone MPI rank. The similar conversion is used in the subsequent wrappers.

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

(iii) **`MPI_Allreduce:`** All teams within a rank carry out a local reduction first and then only the zeroth thread calls the real `MPI_Allreduce` and passes the locally reduced buffer as an input. Once the real `MPI_Allreduce` returns, all teams copy the data from the globally reduced buffer back to their own output buffers. C11 atomic operations are used for busy waiting rather than using any locks.

(iv) **MPI_Iprobe and Improbe:** Each team is assigned with 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 real Improbe is issued and if the message at the head of the MPI queue is destined for the same team, then again `MPI_mecv` is issued. If the incoming message is tagged for another team, then the receiving team inserts the handle in the destination team's queue. The method avoids the blocking of MPI queues when the intended recipient of the MPI queue's head is busy and does not issue probe.

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

(vi) **MPI_Waitall:** A use of global global locks in MPICH `MPI_Waitall` stalls other threads and MPI operations do not progress. Hence a `MPI_Waitall` wrapper was implemented by calling `MPI_Testtall` and busy waiting until `MPI_Testtall` returns true. This method provided about 15–20% speedup over threaded `MPI_Waitall`.

(b) **Parallelizing serial code:** The bottleneck of fork - join was no longer observed after profiling MPI Endpoints. However, this new approach exposed a load imbalance due to serial code. The packing and unpacking of MPI buffers and a local data transfer are executed by the main thread for all the data. Compared to the MPI-only version, the amount of data per rank is "number of threads" times larger, assuming the same workload per core. Thus the serial workload of packing - unpacking for the main thread also increases by "number of threads" times. The solution was to introduce OpenMP pragmas to parallelize the loops associated with these buffers. Thus each buffer could then be processed independently.

(c) **Interface for `parallel_for`:** A downside of explicitly using OpenMP in Hypre is possible incompatibilities with other threading models. in the spirit of [7] an interface was introduced that allows users to pass their own version of "`parallel_for`" as a function pointer during initialization and this user-supplied parallel for is called by simplified BoxLoop macros. Users of Hypre can implement `parallel_for` in any threading model they wish and pass on to Hypre to make flexible.

(d) **Improving auto-vectorization:** The loop iterator macros in Hypre operate using dynamic stride which prevents the compiler from vectorizing these loops. To fix the problem, additional macros were introduced specifically for the unit stride case. The compiler was then able to auto-vectorize some of the loops and gave additional 10 to 20% performance boost depending on the patch size.

3 GPU Hypre Performance Characterization and Profiling

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. 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 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 3 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 500 ms, while the total execution time was 1.6 s. 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 10 µs - the launch overhead of the kernel on V100 (which was shown in the nvprof output).

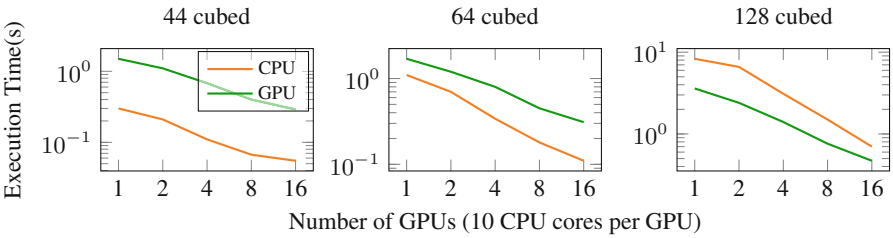


Fig. 3. GPU performance variation based on patch size

Table 2 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

Table 2. Top five longest running kernels before and after merging

Before merging				After merging			
Name	Calls	Tot time	Avg time	Name	Calls	Tot time	Avg time
MatVec	3808	110.69 ms	29.067 us	MatVec	3808	110.59 ms	29.040 us
elax1	2464	55.326 ms	22.453 us	Relax1	2464	55.350 ms	22.463 us
Relax0	2352	45.153 ms	19.197 us	Relax0	2352	44.987 ms	19.126 us
InitComm	20656	38.544 ms	1.8650 us	Axpy	1660	35.664 ms	21.484 us
inComm	20688	37.894 ms	1.8310 us	Memcpy-HtoD	12862	26.689 ms	2.0750 us

average execution time per kernel execution is just 1.8 µs. On the other hand the launch overhead of the kernel on V100 is 10 µs (which was revealed in the profile output). Thus the launch overhead of pack-unpack kernels consumes 0.4 s of 1.6 s (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 2, 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 2. 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 HPCToolkit showed that the MPI wait time for matrix vector multiplication and for relaxation 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 Experiments

4.1 CPU (KNL) Experiments

Choosing the Patch Size: Initial experiments using only the Hypre solve component on a small node count showed the speedups increase with the patch size. Both MPI+OpenMP and MPI EP versions were compared against the MPI only version for different patch sizes. As shown in Table 3, MPI+OpenMP version always performs slower than the MPI Only version, although the performance improves a little as the patch size is increased. On the other hand, the MPI EP model performed nearly as well as the MPI Only version for 16^3 and 32^3 patch sizes on 2 and 4 nodes, but broke down at the end of scaling. With 64^3 patches, however, MPI EP performed up to 1.4x faster than the MPI Only version. As a result, the patch size of 64^3 was 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 works 4x to 8x slower than the MPI Only version. In case of Hypre-MPI EP, the worst case slowdown of 1.8x was experienced for 512 nodes and the fastest execution matched the time of Hypre-MPI Only. This experience together with the results presented above straces 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 the 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 which 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 was chosen for the scaling studies on DOE Theta. The two main components are: (i) Reverse Monte Carlo Ray Tracing (RMCRT) which is used to solve for the radiative-flux divergence during the combustion [9] and (ii) pressure solve which uses Hypre. RMCRT has previously been converted to utilize multi-threaded approach that preforms faster than the MPI only version and also reduces memory utilization [12]. 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 were used for strong scaling experiments on DOE Theta: small (512^3), medium (1024^3) and large (2048^3). The coarser mesh for RMCRT was fixed to 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 was configured in cache-quadrant mode for the experiments. Hypre and Uintah were 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 was launched per core (i.e., 64 ranks per node) while running the MPI only version. For the MPI+OpenMP and MPI EP version, four ranks were launched per node (one per KNL quadrant) with 16 OpenMP threads per rank. The flexibility of choosing team size in MPI EP allowed running the multiple combinations of teams x worker threads within a rank: 16x1, 8x2 and 4x4. The fastest results among these combinations were selected.

4.2 GPU Experiments

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.

5 Results

5.1 KNL Results on Theta:

Table 4 shows the execution time per timestep in seconds for the RMCRT and Hypre solve components on DOE Theta. The multi-threaded execution of RMCRT shows improvements between 2x to 2.5x over the MPI Only version for the small problem and 1.4x to 1.9x for the medium size problem. Furthermore, the RMCRT speedups increase with the scaling. This performance boost is due to the all to all communication needed for the RMCRT algorithm is reduced by 16 times when 16 threads are used per rank. The multi-threaded version also results in up to 4x less memory allocation per node. However, the RMCRT performance improvements are hidden by poor performance of Hypre in the MPI+OpenMP version. As compared to the MPI Only version, a slowdown of 2x can be observed in Hypre MPI+OpenMP in spite of using 64^3 patches. The slowdowns observed are as bad as 8x for smaller patch sizes. Using optimized

Table 4. Theta results: The execution time per timestep in seconds for RMCRT, Hypre and total time up to 512 KNLs.

Nodes	MPI Only			MPI+OpenMP			MPI EP		
	Solve	RMCRT	Total	Solve	RMCRT	Total	Solve	RMCRT	Total
2	36	35	71	76	17	93	24	16	40
4	18	23	41	38	10	48	13	9	22
8	10	18	28	20	7	27	8	7	15
16	40	34	74	80	25	105	32	24	56
32	20	30	50	41	19	60	16	17	33
64	10	29	39	22	15	37	10	15	25
128	42	74	116	83	23	106	36	21	57
256	19	82	101	44	21	65	18	21	39
512	11	72	83	23	20	43	12	22	34

version of Hypre (MPI EP + partial vectorization) not only avoids these slowdowns, but also provides speedups from 1.16x to 1.5x over the MPI Only solve. The only exceptions are 64 nodes and 512 nodes, where there is no extra speedup for Hypre because the scaling breaks down. Because of the faster computation times (as observed in "Solve Time" of Table 4), 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 5 shows the percentage of solve time spent in waiting for the communication. During first two steps of scaling, the communication wait time also scales, but increases during the last step for eight 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.

Table 5. Theta results: communication wait time for MPI EP.

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

As both the components take advantage of the multi-threaded execution, the combination the overall simulation can lead to the combined performance boost of up to 2x as can be observed in the "Total" column of Table 4. It shows how the changes made to Hypre attribute to an overall speedups up to 2x.

5.2 GPU Results on Lassen

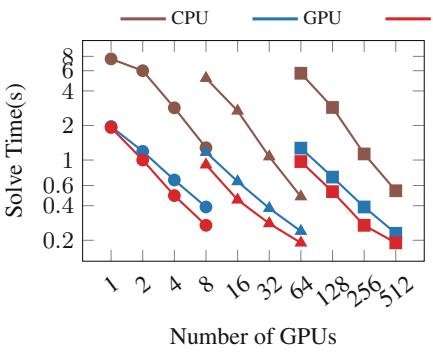


Fig. 4. Strong scaling of solve time

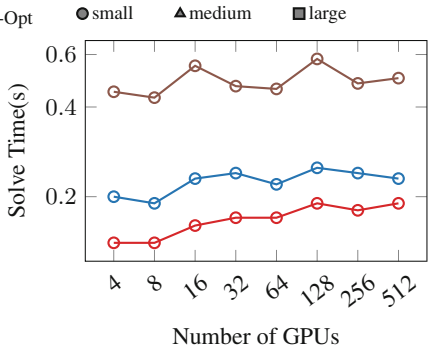


Fig. 5. Weak scaling of solve time

The strong scaling plot in Fig. 4 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 5 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 Fig. 4 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.

6 Conclusions and Future Work

In this paper it has been shown that the MPI-Endpoint approach makes a threaded version of Hypre as fast or faster than the MPI-only version when

used with multiple patches and enough workload. Thus other multi threaded applications which use Hypre could benefit from this approach and achieve overall speedup as demonstrated on Theta. Similarly, improved GPU speedups can help in gaining overall speedups for other Hypre-cuda users.

One of the bottlenecks for the MPI EP version was locks within MPI - especially for smaller patches. This bottleneck can be improved if the lock-free MPI implementations are available or if the End Point functionality [4] is added into the MPI standard. This work used MPI EP to reduce the OpenMP synchronization overhead. However, the EP model can achieve a sweet spot between "one rank per core" and "one rank per node with all cores using OpenMP" and reduce the communication time up to 3x with the minimal OpenMP overhead, which can lead to better strong scaling as shown in [15].

On GPUs the current optimized version shows around 40 to 50% time 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 for the dependent computations after the communication is completed. As these kernels do not have enough work to justify the launch it resulted into slightly slower overall execution times during the initial experiments of overlapping communications. Similar behavior was observed by [1]. 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. This is the work in progress, as is the application of the code to full-scale combustion problems. Another option for speeding up the algorithm is to use communication avoiding approaches e.g., see [10] 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 [13], 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. Ali, Y., Onodera, N., Idomura, Y., Ina, T.: 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), pp. 1–8. IEEE (2019)
2. Baker, A., Falgout, R., Kolev, T., Yang, U.: Scaling hypre's multigrid solvers to 100,000 cores. In: Berry, M., et al. (eds.) High-Performance Scientific Computing, pp. 261–279. Springer, London (2012). https://doi.org/10.1007/978-1-4471-2437-5_13
3. Berzins, M., et al.: Extending the Uintah framework through the petascale modeling of detonation in arrays of high explosive devices. SIAM J. Sci. Comput. **38**, S101–S122 (2016)
4. Dinan, J., et al.: Enabling communication concurrency through flexible MPI endpoints. Int. J. High Perform. Comput. Appl. **28**(4), 390–405 (2014)

5. Falgout, R.D., Jones, J.E., Yang, U.M.: Pursuing scalability for hypre's conceptual interfaces. ACM Trans. Math. Softw. (TOMS) **31**(3), 326–350 (2005)
6. Gahvari, H., Gropp, W., Jordan, K.E., Schulz, M., Yang, U.M.: Modeling the performance of an algebraic multigrid cycle using hybrid MPI/OpenMP. In: 2012 41st International Conference on Parallel Processing, pp. 128–137, September 2012
7. Holmen, J.K., Peterson, B., Berzins, M.: 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) (2019). In conjunction with SC19
8. Humphrey, A., Berzins, M.: An evaluation of an asynchronous task based dataflow approach for Uintah. In: 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC), vol. 2, pp. 652–657, July 2019
9. Humphrey, A., Harman, T., Berzins, M., Smith, P.: A scalable algorithm for radiative heat transfer using reverse Monte Carlo ray tracing. In: Kunkel, J.M., Ludwig, T. (eds.) ISC High Performance 2015. LNCS, vol. 9137, pp. 212–230. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20119-1_16
10. Idomura, Y., et al.: 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), pp. 17–24. IEEE (2018)
11. Kumar, S., et al.: Scalable data management of the Uintah simulation framework for next-generation engineering problems with radiation. In: Yokota, R., Wu, W. (eds.) SCFA 2018. LNCS, vol. 10776, pp. 219–240. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-69953-0_13
12. Peterson, B., et al.: Demonstrating GPU code portability and scalability for radiative heat transfer computations. J. Comput. Sci. **27**, 303–319 (2018)
13. Schmidt, J., Berzins, M., Thornock, J., Saad, T., Sutherland, J.: Large scale parallel solution of incompressible flow problems using Uintah and hypre. SCI Technical report UUSCI-2012-002, SCI Institute, University of Utah (2012)
14. Schmidt, J., Berzins, M., Thornock, J., Saad, T., Sutherland, J.: 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), pp. 458–465 (2013)
15. Sridharan, S., Dinan, J., Kalamkar, D.: Enabling efficient multithreaded MPI communication through a library-based implementation of MPI endpoints. In: International Conference for High Performance Computing, Networking, Storage and Analysis, SC14, pp. 487–498. IEEE (2014)
16. Zambre, R., Chandramowlishwaran, A., Balaji, P.: Scalable communication endpoints for MPI+Threads applications. In: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), pp. 803–812, December 2018