Porting and Evaluation of a Distributed Task-driven Stencil-based Application

Eric Raut  
eric.raut@stonybrook.edu  
Total EP Research & Technology  
Stony Brook University  
Stony Brook, New York, USA

Jonathon Anderson  
janderson@rice.edu  
Total EP Research & Technology  
Rice University  
Houston, Texas, USA

Mauricio Araya-Polo  
Jie Meng  
Total EP Research & Technology  
Houston, Texas, USA

ABSTRACT
Alternative programming models and runtimes are increasing in popularity and maturity. This allows porting and comparing, on competitive grounds, emerging parallel approaches against the traditional MPI-X paradigm. In this work, an implementation of distributed task-based stencil computation is compared with a traditional MPI-X implementation of the same application. The Legion task-based parallel programming system is used as an alternative to MPI, but the underlying OpenMP approach is kept at the subdomain level. Overall results are promising toward making this alternative method competitive to the traditional MPI approach. In future work, extensions to other applications will be explored, as well as the use of GPUs.

CCS CONCEPTS
• Computing methodologies → Parallel programming languages.

KEYWORDS
tasks, stencil, finite difference, wave equation

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ACM Reference Format:

1 INTRODUCTION
The traditional approach to reduce the time-to-solution of explicit and implicit methods for numerical simulation is to parallelize the computing. Computations can be parallelized on a single node using shared-memory parallelism, for which OpenMP is a commonly used technique. When more processing power or memory is desired than a single node can provide, distributed-memory parallelism techniques are used to split the computation among multiple nodes. MPI is a standard approach to distributed-memory parallelism.

These paradigms can be combined: a common approach is known as MPI-X, a multi-layer hybrid approach where a message-based runtime coordinates the communication of data among computing nodes, and X solve the local problem. X can be any shared-memory oriented programming model, possibly including task-based parallelism at the shared-memory level (e.g., OpenMP, Intel TBB, Kokkos), or extended to accelerators, such as CUDA or HIP. A possible alternative is to replace the MPI runtime with a PGAS (partitioned global address space)-based one, which exposes a global address space to each processor, partitioned into separate sections for the memory on each processor.

The contribution of this work is to evaluate an alternative programming model to solve the problem described above. We use the Legion model [7] to handle distributed-memory parallelism. Legion includes an OpenMP implementation, so we retain the use of OpenMP for shared-memory parallelism. It can be summarized as Legion+OpenMP. Furthermore, we evaluate our Legion-based implementation on the newly introduced Stony Brook University’s Ookami cluster, a cluster utilizing the A64FX processor.

Legion is a global task-based programming model based around the concept of “logical regions”, a way of storing data that allows the Legion runtime to automatically extract task-level parallelism from the application. In general, task-based programming has the potential to help alleviate load-balancing issues in applications and free developers from the burden of manual communication that is used in MPI-style programming. Although the mini-application studied in this work is fairly regular, making significant performance improvements from task-based programming unlikely, we expect task-based programming to offer greater benefits over traditional programming techniques as we add different kernels to the application.

The layout of the document is the following: in Section 2 the challenge addressed by our method is introduced. Section 3 reviews related work. Sections 4 and 5 describe MiniMod, the stencil-based application studied in this paper, and its rewrite in C. Section 6 presents our Legion-based implementation of the application. Section 7 describes our evaluation methodology, experimental results, and a discussion of our findings. Section 8 summarizes our conclusions and briefly discusses our plans for future work.
2 PROBLEM DESCRIPTION

The proposed work is motivated by the need to speed up the solution of wave equation-based applications. In this class of applications, most of the time is spent in the computation of the spatial differential operator, discretized as a stencil. The challenge is tackled by introducing parallelism, which in turn brings a new set of considerations.

2.1 Wave propagation and stencil computation

The propagation of waves through a variety of mediums is a well-known phenomena for which established numerical approaches exist. The following governing equation captures the phenomena when the medium is assumed to be isotropic:

\[ \frac{1}{V^2} \frac{\partial^2 u}{\partial t^2} - \nabla^2 u = f, \]  

(1)

where \( u = u(x, y, z) \) is the wavefield, \( V \) is the Earth model (with velocity as rock property), and \( f \) is the source perturbation. An example of a wave propagating is provided in Figure 1.

This PDE can be solved numerically using several methods. For simplicity, we chose the method of Finite Differences (FD) for this work. The time-discretized form of Equation 1 using FD is:

\[ u^{n+1} = Qu^n + u^{n-1} = \left( \Delta t^2 \right) \nabla^2 f^n, \]  

(2)

with \( Q = 2 + \Delta t^2 V^2 \nabla^2 \).

The discretization in space is results in the following:

\[ \nabla^2 u(x, y, z) \approx \sum_{m=1}^{4} c_{xm} [u(i + m, j, k) + u(i - m, j, k)] + c_{ym} [u(i, j + m, k) + u(i, j - m, k)] + c_{zm} [u(i, j, k + m) + u(i, j, k - m)], \]  

(3)

where \( c_{xm}, c_{ym}, c_{zm} \) are discretization parameters. Equation 3 describes a 25-point stencil in 3D space, with four points in each direction as well as the centre point. The memory access pattern of this kind of stencil computation is not friendly to modern hierarchical memory systems: to compute the central stencil point, some neighbors are far in terms of memory (e.g., on different cache lines). This consideration in addition to the use of new hardware platforms make stencil computation an open research challenge. Relevant literature on this topic is covered in Section 3.

2.2 Parallel Stencil Computation

In Figure 2 left panel, it can be observed that each grid point can be computed in parallel provided each element is able to access their neighboring values before an in-place update, in order to retain the data dependencies imposed by Eq. (3). Thus given a sufficient number of computation units, solving this equation can be achieved in straight forward parallel fashion where all elements are computed at once. To achieve this in practice requires a coherent memory address space shared across a very large number of computation units. No current hardware platform is able to fulfill these requirements for moderately sized domains (1024³–2048³, roughly 1–8 billion elements), the closest are provided by GPGPU systems (such as the NVIDIA V100 with 5140 processing units). A more suitable alternative is to divide the domain into subdomains, which can be mapped to individual processing units and computed in parallel. A simple example of these where three threads are mapped to corresponding rows of a compute domain is depicted in Fig. 2 left panel. This is usually implemented using runtimes such as OpenMP on CPUs and languages such as NVIDIA’s CUDA or AMD’s HIP for GPUs.

When scaling to large problems, a single compute node may not be sufficient in either memory capacity or processing throughput. The solution is to use a distributed parallel approach. In this case memory is distributed in multiple locations without coherent access, thus communication between the compute nodes is needed to accurately perform these stencil computations. For instance, as shown in Figure 2, for a stencil centered at \((x_0, y_0)\) of \(D_0\), its neighbors \((x_0, y_0+1)\) and \((x_0, y_0+2)\) lay within \(D_1\), requiring the transfer of data between these two subdomains. This approach is widely used and frequently implemented using runtimes such as MPI or PGAS, the main difference between the two being the use of explicit or implicit data motion respectively.

Finally, these two classes of approaches can be combined to form hybrid parallel approaches, for instance of the form MPI+OpenMP or MPI+CUDA. Sections 4 and 5 describe our implementation of these hybrid approaches for the research presented here.

3 RELATED WORK

3.1 Stencil computation

New hardware technologies (such as GPUs, FPGA, etc) have motivated researchers to further investigate how to re-purpose stencil algorithms to take advantage of their unique characteristics, following two major approaches: programming models and pure algorithmic optimization. On the programming models front, in 2012, Ghosh et al. [17] analyzed the performance and programmability of three high-level directive-based GPU programming models (PGI, CAPS, and OpenACC) on an NVIDIA GPU on kernels for reverse time migration (RTM). In 2017, Qawasmeh et al. [32] implemented an MPI + OpenACC approach for seismic modeling and RTM. In a separate
Figure 2: A 9-point 2D stencil is shown, where green colored elements represent grid points within the computing domain, yellow elements are not computed but needed by the stencil (e.g. to impose boundary conditions). In the left panel, a simple threads to rows mapping is presented. In the right panel, two domains ($D_0, D_1$) shaded light blue and light orange are depicted.

The hard boundary implies no memory coherency, therefore information must be exchanged between the two domains in order to solve Equation 3 correctly.

line of research, domain-specific languages (DSLs) and domain-specific parallel programming models and compiler optimizations for stencils have been proposed (e.g., [19, 23, 34]). Performance models have been developed for this computing pattern (see [11]), and the kernel has been ported to a variety of platforms ([3, 4]). Most recently, Sai et al. [37] studied high-order stencils with a manually crafted collection of implementations of a 25-point seismic modeling stencil in CUDA for the latest GPU hardware. Along this line of hardware-oriented stencil optimization, Matsumura et al. ([24]) proposed a framework (AN5D) for GPU stencil optimization, obtaining remarkable results.

On the algorithmic optimization path, spatial and temporal blocking has been proposed ([16, 42]). A further example is the semi-stencil algorithm proposed by De la Cruz et al. [12], which offers an improved memory access pattern and data reuse. Promising results are also provided by a higher dimension cache optimization introduced by Nguyen et al. [28] which accommodates both thread-level and data-level parallelism.

3.2 Task parallelism
Task-based parallel programming has been recognized as a promising approach to improve performance in scientific applications. For example, in [27], Moustafa et al. illustrated the design and implementation of a FD method-based seismic wave propagation simulator using PaRSEC, although their implementations are limited by scalability and only implemented on CPUs. OpenMP version 3.0 introduced support for task parallelism. Several researchers have implemented application codes and benchmarks in OpenMP tasks and evaluated the performance [5, 15, 36, 40, 41]. Recently, Raut et al. [33] implemented OpenMP task parallelism into the industrial seismic modeling kernel used in the current paper, and presented taskified kernels which are competitive with traditional OpenMP-augmented loops.

Tasks are supported by several programming systems, including both shared-memory (e.g., OpenMP) and distributed-memory programming models. For example, Cilk [8] is an early shared-memory programming API supporting tasks using spawn keyword. Intel Thread Building Blocks [35] and Kokkos [10] also support shared-memory task parallelism. StarSs [31] is a shared-memory task-based framework using a pragma syntax. OmpSs [14] is an attempt to extend OpenMP with tasking features using StarSs runtime.

3.3 Distributed task parallelism
Distributed-memory task-based systems have been explored as well, in which the runtime automatically schedules tasks among the available nodes and implicitly handles communication and data transfer. Charm++ [2] is a C++ framework supporting distributed task parallelism. PaRSEC [9] enables an application to be expressed as a “parameterized task graph” which is problem-size-independent and therefore highly scalable. HPX [20] is a task-based framework which uses a global address space to distribute computations across nodes. XcalableMP [22] is a PGAS language with elementary support for task parallelism. YML [13, 18] allows the user to specify a computation as a graph of large-scale tasks and supports combination with XcalableMP. StarPU [6] supports OpenMP-style pragmas and provides a runtime for distributed execution. Klinkenberg et al. [21] propose a framework for distributing tasks across MPI ranks in MPI+OpenMP hybrid applications.

Legion [7] and its DSL Regent [38] are data-centric task-based programming systems developed at Stanford. Soi et al. [39] implement a task-based implicitly parallel meshfree solver using the Regent programming language. Their implementation takes advantage of the optimizations available in Regent and Legion for improving the scalability and performance of the target application, such as index launches and dynamic control replication. The performance of the Regent-based implementation is compared with an MPI-based solution as well as a Julia-based solution, evaluated using both single-node and multi-node CPU systems. Their Regent-based implementation shows a significant degradation in scalability compared to the MPI version, even at relatively small node counts.

4 DISTRIBUTED MINIMOD (DMINIMOD)

MiniMod is a proxy application for seismic modeling, which simulates the propagation of waves through the Earth. It models the propagation of acoustic waves emitted from the source by solving
a Finite Difference (FD) discretized form of the wave equation. The software suite is designed and developed by Total Exploration and Production Research and Technologies [26]. Minimod extracts the core stencil computations used in industrial production simulations. The proxy application purpose is to benchmark new HW platforms for technology tracking. MiniMod is currently not publicly available; however, the plan is to eventually make it available to the community as open-source software.

**dMiniMod** is part of the application suite which is implemented with explicit parallel programming using MPI for running on distributed computer systems. It uses non-blocking communication in MPI to achieve higher performance by overlapping communication and computation. It carries out domain decomposition to partition the global geographical area into several sub-domains so that each process handles one sub-domain independently for fully parallelized computation.

**dMiniMod** provides both a sequential implementation and a hybrid-parallel implementation using the combination of MPI+OpenMP programming models. The hybrid MPI+OpenMP version allows us to further improve system performance by exploiting additional layers of parallelism and utilizing higher numbers of processors past where the scalability of the pure MPI version reaches its limits. The implementation of MPI+OpenMP follows the classical master-only style where all MPI calls are made only from the OpenMP master thread outside of any parallel regions. The hybrid implementation in **dMiniMod** accommodates a clear separation between MPI and OpenMP levels of parallelism.

**dMiniMod** is self-contained and designed to be portable across multiple compilers. In this paper we focus on evaluating one of the kernels contained in Minimod: the acoustic isotropic propagator (high-order spatial stencil) in a constant-density medium [3]. While the kernel itself has not been carefully manually optimized, the goal in this paper is to evaluate the relative scaling of different parallel programming models.

## 5 ASF: ADVANCED SEISMIC FRAMEWORK

While **MiniMod** and **dMiniMod** have been proven as useful tools for benchmarking emerging technologies, many of the details that need to be present in scientific applications are elided, such as saving results to disk for further analysis. This puts a burden on the application developer to reproduce the algorithm and performance present in the original benchmark, while making alterations to be suitable for the application’s needs. Similarly, as a benchmark suite, every combination of technologies is implemented as a separate port of the original code, so extending **MiniMod** to support a new technology “X” requires a substantial and cumulatively increasing effort. To prevent these from becoming serious issues as we continued development, we first spent the time to re-implement **MiniMod** suite as a library suitable for use by scientific applications.

The resulting library **Advanced Seismic Framework** (ASF) handles the myriad of technology combinations by subdividing an implementation into discrete components, responsible for computation, communication, and external I/O. Each component supports multiple implementations that can be combined and at runtime at will, providing functionality extending beyond **MiniMod** with a much reduced codebase. Applications manage the abstract transfer of data between components via opaque handles, and implement the main processing loop using asynchronous operations to support slower high-level languages such as Python.

The initial version of ASF included one representative kernel from **MiniMod**, namely, **acoustic_iso_cd** [26]. It included a single-threaded computation backend, as well as parallelized computation backends using OpenMP and CUDA. Figure 4 presents some initial performance results from a simple ASF benchmark compared against the nearest equivalent in the original **MiniMod** and **dMiniMod** suite, showing that the ASF framework does not significantly inhibit the performance of the contained implementations.

ASF also provides the option to enable distributed implementation with MPI for collective operations. The number of subdivisions in each dimension for the domain decomposition can be configured by the user. Figure 3 presents the trace view of running ASF with four MPI ranks on a grid sizes of $1024^3$. The number of domains are $(1, 2, 2)$ in $(x, y, z)$ dimensions respectively. The traces and statistics are collected using the HPCToolkit [25] from Rice University. The profiling results show that the majority of the runtime is consumed by kernel computations, while the MPI communication only cost 2.6% of the overall simulation time.

## 6 LEGION IMPLEMENTATION

This section describes our implementation of ASF in the Legion programming model, which we call “ASF-Legion”. The ASF-Legion implementation performs computations on CPUs; it does not include accelerators.

Computation in Legion is done by tasks. For each task, the application developer specifies the logical regions as inputs or outputs. The Legion runtime then determines which tasks can run at the same time, and schedules them for execution on the available processors.

Algorithm 1 gives a high-level overview of the main solver in **Minimod** and **ASF** applications. In the Legion implementation, the wavefield solution step (lines 3-5) is implemented as a task, and one task is launched for each subregion of the domain. Initially, the source update step (line 6) was a separate task, but merging it into the wavefield update task improved performance due to the small execution time of the source update task.

```plaintext
Data: f: source
Result: u^n: wavefield at timestep n, for n ← 1 to T
1 u^0 := 0;
2 for n ← 1 to T do
3     for each point in wavefield u^n do
4         Solve Eq. 2 (left hand side) for wavefield u^n;
5     end
6     u^n = u^n + f^n (Eq. 2 right hand side);
7 end

Algorithm 1: Minimod and ASF high-level description [33]
```

### 6.1 Parallelism in ASF-Legion

ASF-Legion contains two levels of parallelism: Legion tasks and OpenMP threads. First, the domain is split into blocks, and the
Figure 3: Traces of running ASF with 4 MPI ranks from the Traceviewer of HPCToolkit [25]: the main panel on the top left presents sequences of samples of each trace line rendered, while each of the four horizontal sub-panels represent the trace-view from one of the four MPI ranks. The different colors represent the time spent on different subroutines which are listed on the right panel.

Figure 4: Comparison of performance on a single Summit node between ASF (in blue) and the nearest equivalent algorithm from the MiniMod suite (in red), simulating 1000 timesteps at various grid sizes. Lower is better.

Legion runtime distributes these blocks among the Legion tasks. The second level, within each task, is OpenMP parallelism, where OpenMP pragmas are used to distribute iterations along the largest-stride dimension to threads. In order to avoid conflict between Legion’s threads and OpenMP threads, Legion implements its own OpenMP runtime. In ASF-Legion, blocks of the domain are assigned to a Legion task, which has its own OpenMP runtime and several threads assigned to it. OpenMP parallelism is then used within the task. Figure 5 is a diagram showing the different levels of parallelism.

6.2 Domain Decomposition and Mapping

The data needed by the kernel is stored as five logical regions: the wavefield at the last and current timesteps, the density field (which is constant for this kernel), and two PML wavefield arrays.

In Legion, partitions are crucial for expressing the independence of subtasks that can run in parallel. Logical regions are partitioned into subregions that are available for simultaneous read/write access. In this application, the domain is split into equally-sized pieces in each of the three dimensions. We declare two partitioning schemes for the 3-D logical regions: a disjoint set of partitions that creates equally sized blocks that are divided among the tasks, and a second set of partitions that are the same size as the first set but also includes the ghost points of each block. This dual partition-scheme setup allows us to concisely express the inputs and outputs of each task for the wavefield solution step.

This blocked partitioning paradigm is a common use case in Legion, and can be expressed concisely using the Legion utility create_partition_by_restriction. Figure 6 shows an example. The number of partitions in each dimension is configurable.

After partitioning the domain, at each timestep, one task is launched per subregion. Figure 7 shows how tasks are launched in Legion. By using an index task launch, rather than launching each subtask individually, the Legion runtime is able to amortize the overhead of the task launches. Unlike the MPI version, in Legion there need not be a 1-to-1 mapping between Legion tasks and processors within a timestep. Nonetheless, in this study, one Legion task is used per process, analogous to the distribution of one block per MPI rank.

The Legion programming model allows the user to write a custom mapper to specify how tasks and data should be mapped to
Figure 6: Code showing how domain is partitioned. \texttt{nb(x,y,z)} refer to the number of blocks in each dimension. Each point in the color space \texttt{(color\_is)} is multiplied by the 3 x 3 transform matrix to generate an offset. This offset is added to the extent.

```c
IndexSpace is = runtime->create_index_space(ctx, 
    Rect<3>(Point<3>(0,0,0), Point<3>(nx-1,ny-1,nz-1));
IndexSpace color_is = runtime->create_index_space(ctx, 
    Rect<3>(Point<3>(0,0,0), Point<3>(nx-1,ny-1,nz-1));
Transform<3,3> transform;
int bdx = nx/nbx, bdy = ny/nby, bsz = nz/bsz;
// Diagonal transform matrix
transform[0][0] = bdx;
transform[1][1] = bdy;
transform[2][2] = bsz;
Rect<3> extent(Point<3>(0,0,0), Point<3>(nx-1,ny-1,nz-1));
ip = runtime->create_partition_by_restriction(ctx, is, color_is, transform, extent);
```

Figure 7: Code showing how kernel computation tasks are launched. In this example, the logical partition for the waves \texttt{(lp\_waves)} is added to the task as a region requirement, so that one subtask will be launched for each partition of the domain. The \texttt{execute\_index\_space} method schedules the execution of each subtask.

```c
IndexTaskLauncher launcher(KERNEL_TASK, color_is, 
    TaskArgument(argc.sizeof(+arg)), arg_map);
launcher.add_region_requirement(RegionRequirement(
    lp_waves, 0, READ_ONLY, EXCLUSIVE, lr_waves).
    add_field(fid_wave_now));
// add other regions ...
runtime->execute_index_space(ctx, launcher);
```

6.3 Memory Management

Data from logical regions in Legion is accessed using "accessors". When implementing the main computational kernel in ASF-Legion, we noticed that accessing data using an accessor resulted in a substantial slowdown with the Clang compiler — roughly 2-3X for a single thread. After investigation, we determined that this was due to the way the accessor addresses elements in an array. Because the accessor is written generically, each subscript in an array access is multiplied by a variable containing the stride in that dimension. Arrays are stored in row-major order, so the last dimension is stride-1 access. The lack of compile-time knowledge of stride-1 access inhibits an optimization performed by the Clang compiler. We modified the accessor to assume stride-1 access in the last dimension, and recovered the single-thread performance of the original code without Legion.

Table 1: Hardware and software configuration of the experimental platforms. * IBM Spectrum MPI (SMPI)

<table>
<thead>
<tr>
<th>System</th>
<th>Hardware Specs</th>
<th>Software</th>
</tr>
</thead>
<tbody>
<tr>
<td>Summit</td>
<td>CPU cores 2xIBM Power9</td>
<td>LLVM 9.0</td>
</tr>
<tr>
<td></td>
<td>Memory 512 GB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L3 10 MB (x2 cores)</td>
<td>SMPI * 10.3.1</td>
</tr>
<tr>
<td></td>
<td>L2 512 KB (x2 cores)</td>
<td>GASNet 2020.3.0</td>
</tr>
<tr>
<td></td>
<td>L1 32+32 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lithography 14nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interconnect InfiniBand EDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDP 190x2 W (only CPUs)</td>
<td></td>
</tr>
<tr>
<td>Ookami</td>
<td>CPU Fujitsu A64FX</td>
<td>GCC 10.2.1</td>
</tr>
<tr>
<td></td>
<td>CPU cores 12x4</td>
<td>OpenMPI 4.0.5</td>
</tr>
<tr>
<td></td>
<td>Memory 32 GB</td>
<td>GASNet 2020.3.0</td>
</tr>
<tr>
<td></td>
<td>L2 8 MB (x12 cores)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>L1 64+64 KB</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lithography 7nm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interconnect InfiniBand HDR</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TDP 160 W (full node)</td>
<td></td>
</tr>
</tbody>
</table>

7 EXPERIMENTAL RESULTS

In this section, the performance results of the experiments conducted with ASF-MPI and ASF-Legion are presented. First, the experimental setup is described, second weak and strong scaling performance of each parallel implementation is introduced. Finally, profiling and analysis are provided.

7.1 Experimental Setup and Platforms

We perform experiments on the Summit computer at Oak Ridge National Laboratory, and the Ookami cluster at Stony Brook University.

Each node of Summit [29] has two sockets containing IBM POWER9 processors, with 44 total CPU cores per node, and 512 GB memory per node. Each node also has 6 NVIDIA V100 GPUs; however, we do not use GPUs in this study.

The Ookami cluster [1] at Stony Brook University is a pre-production system containing the first deployment of the Fujitsu A64FX [30] microprocessor outside of Japan. Each node of Ookami includes the A64FX processor, which has 48 compute cores and 32 GB of HBM2 memory, organized into four NUMA "core memory groups (CMGs)" with 12 cores and 8 GB memory each. Both systems utilize an InfiniBand interconnect with a fat-tree topology.

Detailed information about each system, including the software stack used, is shown in Table 1.

On each Summit node, two Legion or MPI processes are used — one for each socket. Legion reserves some cores to help manage mapping and communication. We use 36 cores on each node (18 per socket) to run OpenMP threads (to perform kernel computations), and the rest are reserved for Legion. To match this configuration, we also use 36 cores per node for ASF-MPI. Although we only use one thread per physical core, we enable hyperthreading (using \texttt{smt4} mode) because in experiments we found that this mode gives a
Table 2: Weak scaling: throughput (stable around the reference is ideal) with 1-32 nodes; grid size $1024^3/16$ per node; on Summit.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>17.9</td>
<td>20.0</td>
<td>100.0%, 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>18.7</td>
<td>19.6</td>
<td>95.6%, 102.0%</td>
</tr>
<tr>
<td>4</td>
<td>18.7</td>
<td>19.7</td>
<td>95.3%, 101.7%</td>
</tr>
<tr>
<td>8</td>
<td>18.8</td>
<td>19.8</td>
<td>95.1%, 100.9%</td>
</tr>
<tr>
<td>16</td>
<td>18.8</td>
<td>19.9</td>
<td>94.9%, 100.6%</td>
</tr>
<tr>
<td>32</td>
<td>18.8</td>
<td>21.1</td>
<td>94.9%, 94.7%</td>
</tr>
</tbody>
</table>

Table 3: Weak scaling: throughput (stable around the reference is ideal) with 1-32 nodes; grid size $1024^3/16$ per node; on Ookami.

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>20.4</td>
<td>22.3</td>
<td>100.0%, 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>20.7</td>
<td>24.0</td>
<td>99.0%, 93.0%</td>
</tr>
<tr>
<td>4</td>
<td>20.6</td>
<td>24.6</td>
<td>99.2%, 90.8%</td>
</tr>
<tr>
<td>8</td>
<td>20.7</td>
<td>25.7</td>
<td>98.7%, 87.0%</td>
</tr>
<tr>
<td>16</td>
<td>20.6</td>
<td>47.9</td>
<td>99.3%, 46.6%</td>
</tr>
<tr>
<td>32</td>
<td>21.1</td>
<td>-</td>
<td>97.1%, -</td>
</tr>
</tbody>
</table>

Table 4: Sensitivity of ASF-MPI and ASF-Legion to the block partitioning scheme (number of partitions in x,y,z dimensions) of the domain. Elapsed time (seconds) is shown for each scheme. These simulations use a grid size of $1024^3$, with 16 nodes, on Summit.

<table>
<thead>
<tr>
<th>Partitioning</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>32:1:1</td>
<td>29.9</td>
<td>25.0</td>
</tr>
<tr>
<td>16:2:1</td>
<td>24.4</td>
<td>22.8</td>
</tr>
<tr>
<td>2:16:1</td>
<td>21.9</td>
<td>41.5</td>
</tr>
<tr>
<td>1:32:1</td>
<td>25.9</td>
<td>54.0</td>
</tr>
<tr>
<td>4:4:2</td>
<td>19.7</td>
<td>142.4</td>
</tr>
</tbody>
</table>

Table 5: Strong scaling: elapsed time for grid size $1024^3$ and corresponding parallel efficiency; Summit

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%] ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>232.7</td>
<td>241.3</td>
<td>100.0%, 100.0%</td>
</tr>
<tr>
<td>2</td>
<td>121.1</td>
<td>122.3</td>
<td>96.1%, 98.6%</td>
</tr>
<tr>
<td>4</td>
<td>64.8</td>
<td>64.5</td>
<td>89.8%, 93.5%</td>
</tr>
<tr>
<td>8</td>
<td>35.3</td>
<td>36.1</td>
<td>82.4%, 83.6%</td>
</tr>
<tr>
<td>16</td>
<td>19.7</td>
<td>22.9</td>
<td>73.8%, 65.8%</td>
</tr>
</tbody>
</table>

7.2 Scalability Results and Analysis

We begin by evaluating the weak scaling of ASF-MPI and ASF-Legion. Weak scaling is evaluated with a simulation grid size of $1024^3/16$ grid points per node, from 1 to 32 nodes. The grid distribution for each node number is chosen as the per-rank “best” distribution for 16 nodes (e.g., Table 4), expanded along the x-axis for each additional rank. For example, on Summit, the best Legion distribution for 16 nodes is 16:2:1. Therefore, the Legion weak-scaling experiments on Summit use a grid size of $64:512:1024$ per rank, for a total grid size of $(64\times\text{nranks})\times512:1024$, where there are two ranks per node.

The expectation with weak scaling performance is that as long as more resources are deployed and the problem size increased proportionally, the parallel efficiency should remain the same, or close to the reference. In order to have a meaningful reference, the size of the problems should be demanding enough to stress all relevant system resources. The results introduced in Table 2 (Summit) indeed show the expected behavior for both implementations. In Table 3 (Ookami), results for ASF-Legion are less promising after 8 nodes; in the extreme, the 32 node case failed. The performance of ASF-Legion on Ookami is not up to what is achieved on Summit or ASF-MPI on the same cluster, the cause of this discrepancy in under investigation.

Table 4 shows the sensitivity of ASF-MPI and ASF-Legion to how the domain is partitioned into blocks, for selected partition schemes. In general, ASF-Legion is much more sensitive to the partition scheme than ASF-MPI, with some configurations performing much better than others. In the strong-scaling results below, the best partition scheme is selected for each grid size and number of nodes.

In terms of strong scaling, both implementations follow on Summit the same trends for the $1024^3$ and $2048^3$ grids, as can be seen in Table 5 and Table 6. The parallel efficiency drops after the 8 nodes for the $1024^3$ case (Table 5). On Summit, for the 16 node case, each rank is solving a $320^3$ domain, where the halo exchange among neighbors starts to dent the overall performance. In Table 6 the same patterns are observed but the drop for 16 node case is less pronounced since a much larger problem is being solved.
(≈ 512³ for ASF-Legion), giving more chances for computing and communication overlap.

The ASF-Legion implementation is ≈ 20 seconds slower than ASF-MPI on Summit for 16 nodes (2048³ grid, Table 6); this represents ≈ 16% of slowdown, which is within the variation of the results when different domain decomposition are considered (see Table 4). The same trend is observed in Table 7, up to when 8 nodes are considered, in which ASF-MPI outperforms ASF-Legion by 7.6 seconds which corresponds to ≈ 19%. Besides the performance drop associated with the 16 node case, for which further analysis is underway, the ASF-Legion performance is comparable with the one reached by ASF-MPI, which is remarkable for such a regular computing pattern, with—in this case—a low number of opportunities to exploit task-oriented parallelism given the reduced number of computing kernels.

### 7.3 Profiling

Legion includes a built-in profiler to aid the developer in optimizing performance of applications. The profiler shows a time trace of task execution and memory copies during the simulation. During the development of this application, Legion’s built-in profiler was critical to diagnosing the performance issues and improving the code.

Figure 10 shows example time traces generated by the profiler. The profile shows occasional large gaps between stencil computations for successive timesteps (more so on Ookami). These gaps may indicate overhead associated with the Legion runtime on Ookami, but also suggest that there are opportunities for overlapping communication and computation for improved performance.

### 8 CONCLUSION

ASF-Legion achieved competitive results against ASF-MPI, in both raw performance and scalability. Even though this application is relatively regular, and therefore has less potential for improvement over MPI, it is promising that the task-based approach yields results comparable to the MPI approach in performance. In particular, results are robust on the production system, Summit. On Ookami, which is currently a pre-production system, there are some performance issues, particularly affecting Legion, that need to be addressed.

Implementing ASF from a task-based perspective was relatively straightforward, although intrusive. Legion can’t act as a drop-in replacement for MPI, due to needing to manage the memory of the application as logical regions, as well as needing to write the application from a global view for task-based parallelism. Nonetheless, the modular structure of ASF (see Section 5) allowed reuse of much of the code of the ASF-MPI application code.

We will continue to investigate performance issues in ASF-Legion. Closing the gap in performance between ASF-Legion and ASF-MPI is a particular goal. We would also like to explore whether ASF-Legion can even gain an edge over ASF-MPI in performance. Over-decomposition, which would expose more task parallelism to the Legion runtime, is one possibility to achieve this.

Future work will include using GPUs to offload the most demanding computing sections, and to explore combining GPU kernels with task-based parallelism at the node-level. Also, other kernels will be implemented, such as IO and other physics-oriented processes.

### ACKNOWLEDGEMENTS

We would like to thank Total E&P Research and Technology USA for their support and permission to share this work. We also thank the Institute of Advanced Computational Science at Stony Brook University and ORNL for the accesses of computing resources (Ookami and Summit, respectively). Ookami is funded by NSF grant #1927880. Finally, we thank the Legion developers for helpful discussions and insight regarding the performance of ASF-Legion.

### REFERENCES


---

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%]</th>
<th>ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1694.4</td>
<td>1694.4</td>
<td>100.0%, 100.0%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>875.8</td>
<td>875.3</td>
<td>96.7%, 96.8%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>450.8</td>
<td>460.4</td>
<td>94.0%, 92.0%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>233.2</td>
<td>237.0</td>
<td>90.8%, 89.4%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>121.5</td>
<td>141.8</td>
<td>87.1%, 74.7%</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Strong scaling: elapsed time for grid size 2048³; Summit

<table>
<thead>
<tr>
<th>Nodes</th>
<th>ASF-MPI [s]</th>
<th>ASF-Legion [s]</th>
<th>Parallel Efficiency [%]</th>
<th>ASF-MPI, ASF-Legion</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>227.2</td>
<td>221.5</td>
<td>100.0%, 100.0%</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>115.6</td>
<td>148.8</td>
<td>98.3%, 74.4%</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>63.5</td>
<td>76.4</td>
<td>89.4%, 72.4%</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38.5</td>
<td>46.0</td>
<td>73.9%, 60.2%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>22.5</td>
<td>39.6</td>
<td>63.1%, 35.0%</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Strong scaling: elapsed time for grid size 1024³ and corresponding parallel efficiency; Ookami
Figure 8: Throughput and speedup, grid size $1024^3$, on Summit and Ookami. Higher is better. Memory footprint 20GB (with a single node), data precision is 32-bit float.

Figure 9: Throughput and speedup, grid size $2048^3$, on Summit. Higher is better. Memory footprint 160GB, data precision is 32-bit float. When running the experiments in Summit, each MPI rank is mapped to socket, therefore for the largest number of nodes experiments one have 32 subdomains, which means 10GB footprint per socket (4% of the available memory per socket).

Figure 10: Sample extracted from Legion profiler time trace with 16 nodes on Summit (left) and Ookami (right). Each time period is roughly 55 milliseconds. "OpenMP Proc 2" in each trace shows one of the compute processes, and each rectangle to the right of "OpenMP Proc 2" is a task computing the stencil on the subdomain of the grid assigned to that processor. Each task roughly corresponds to one timestep, with the other processors (not shown here) computing the other subdomains of the grid during the same period. The "[x] sys to [y] sys" rows show halo exchange between the indicated processes.