MPI-based Remote OpenMP Offloading: A More Efficient and Easy-to-use Implementation

Baodi Shan
baodi.shan@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

Mauricio Araya-Polo
TotalEnergies EP Research & Technology US
Houston, Texas, USA

Abid M. Malik
amalik@bnl.gov
Brookhaven National Laboratory
Upton, New York, USA

Barbara Chapman
barbara.chapman@stonybrook.edu
Stony Brook University
Stony Brook, New York, USA

ABSTRACT

MPI+X is the most popular hybrid programming model for distributed computation on modern heterogeneous HPC systems. Nonetheless, for simplicity, HPC developers ideally would like to implement multi-node distributed parallel computing through a single coherent programming model. As de facto standard for parallel programming, OpenMP has been one of the most influential programming models in parallel computing. Recent work has proven that the OpenMP target offloading model could be used to program distributed accelerator-based HPC systems with marginal changes to the application. However, the UCX-based version of remote OpenMP offloading still has many limitations in terms of performance overhead and ease of use of the plugin.

In this work, we have implemented a new MPI-based remote OpenMP offloading plugin. By comparing it with the UCX-based version, the new MPI-based plugin has been significantly improved in terms of performance, scalability, and ease of use. Evaluation of our work is conducted using one proxy-app, *XSBench* and an industrial-level seismic modeling code, *Minimod*. Results show that, compared to the optimized UCX-based version, our optimizations can reduce offloading latency by up to 70%, and raise application parallel efficiency by 68% when running with 16 GPUs on data-bound applications. In particular, the introduction of the concept of locality-aware offloading gives developers of HPC programs greater possibilities to take full advantage of modern hierarchical heterogeneous computing devices.

CCS CONCEPTS

• **Computing methodologies** → **Parallel programming languages**.

KEYWORDS

OpenMP, GPGPU, distributed computing

ACM Reference Format:
Baodi Shan, Mauricio Araya-Polo, Abid M. Malik, and Barbara Chapman. 2023. MPI-based Remote OpenMP Offloading: A More Efficient and Easy-to-use Implementation. In *The 14th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'23), February 25, 2023, Montreal, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3448290.3448559

PMAM'23, February 25, 2023, Montreal, Canada

ACM ISBN 978-1-4503-8348-6/21/02…$15.00
https://doi.org/10.1145/3448290.3448559

1 INTRODUCTION

Heterogeneous computing is becoming one of the hottest topics in High Performance Computing (HPC). In heterogeneous computing, GPU acceleration is the practice of using a graphics processing unit (GPU) in addition to a central processing unit (CPU) to speed up processing-intensive operations. At the same time, the diversity of device types brought by heterogeneous computing makes the portability and maintainability of programs more and more critical.

Since most distributed computing models are designed to execute code only on the CPU, developers of HPC programs often need to write code based on vendor-specific APIs when utilizing heterogeneous acceleration devices, such as NVIDIA's CUDA, AMD's HIP, and Intel's oneAPI. In this case, the poor portability of the program imposes heavy burden on the developers of HPC programs.

OpenMP [14] is the *de facto* HPC programming model for shared-memory parallelism. In OpenMP version 4.0, OpenMP Target allows running a single code on multi-platform heterogeneous devices. In particular, the rich set of heterogeneous devices supported by LLVM/OpenMP gives developers great convenience. Currently, the LLVM/OpenMP Target plugin support includes ARM, AMDGCN, CUDA, PPC, Remote, VE, X86.

Another problem posed by the distributed model is the migration of single-node program implementations. Typically, developers use the MPI+X strategy to migrate programs from intra-node to inter-node implementations. This means that developers need to learn specifically how to work across nodes and how to synchronize. Therefore, how to replace MPI+X to reduce the workload of HPC program development has also received attention.

Recent work of remote OpenMP offloading based on gRPC and UCX by Patel and Doerfert [15] has shown that through extensions in the LLVM/OpenMP runtime system, specification conforming

OpenMP offloading applications can seamlessly utilize accelerators attached to remote compute nodes. This shows the potential for transforming OpenMP into an all-encompassing programming model for writing performance portable and maintainable scientific applications in the era of heterogeneous Supercomputing as an alternative to hybrid programming models like MPI+X. Lu et al. [10] present a detailed analysis of the performance and overhead of the remote OpenMP offloading plugin and release an optimized version. However, according to the overhead analysis, even the optimized UCX-based remote OpenMP offloading plugin still has significant limitations and shortcomings.

Therefore, we refactored the existing remote OpenMP offloading plugin using MPI as the communication underlying layer. We have compared the new version of the MPI-based plugin with the UCX-based one; then, we analyzed the porting and performance of the industrial program Minimod with the new version of the plugin. Thus, this work has the following main contributions:

- Refactoring the remote OpenMP offloading plugin using MPI as the underlying communication layer
- Improve the performance of the remote OpenMP offloading plugin with multiple optimizations in the MPI-based version
- Propose the concept of locality-aware offloading, which allows developers better leverage the performance of modern heterogeneous computing devices
- An evaluation of the optimized plugin using proxy-apps of real-world HPC applications
- Propose directions and possible solutions for further optimization of the plugin by analyzing the industry application **Minimod** code and evaluation results

The paper is organized as follows: section 2 describes LLVM/OpenMP remote offloading and the related work,section 3 introduces the implementation of the MPI-based version remote OpenMP offloading plugin, section 4 present improvement brought by the new version, evaluation of our work is described in section 5, and finally section 6 concludes the work and talk about future directions.

2 BACKGROUND

In this section, we describe the details of how an OpenMP target plugin works in LLVM, how the remote OpenMP plugin works in LLVM, as well as related work about the technologies and cases used later in the experimental section.

2.1 OpenMP Offloading in LLVM/OpenMP

Figure 1: LLVM/OpenMP device offloading workflow for CUDA-able devices

An OpenMP application's offloading directives are lowered to functions calls into the host runtime (libomptarget.so), which is the entry point of all offloading operations in the LLVM/OpenMP offloading (workflow shown in Figure 1). The host runtime is target-agnostic, and it loads the device runtime plugin according to the type of device code embedded in the fat application binary. For example, it can load the CUDA plugin libomptarget.rtl.cuda.so, which makes CUDA calls to transfer data and launch kernels.

The host runtime interacts with the device runtime through the device plugin interface, which is a small set of target-agnostic C functions, to perform offloading operations (e.g. __tgt_rtl_data_alloc()). All device runtime plugins must implement this interface, hiding low-level details from the host runtime.

2.2 The UCX-based Remote Offloading Plugin and its Optimizations

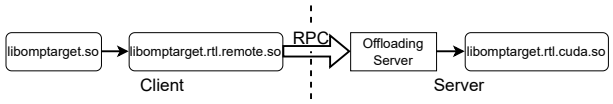

Figure 2: LLVM/OpenMP remote offloading workflow for CUDA devices

The idea of the remote offloading plugin is to use a remote procedure call (RPC) to forward the plugin interface calls to a remote process, so the application can transparently access devices on other machines. The plugin implements a core set of device plugin interfaces that interact with the host runtime. The OpenMP application is compiled as usual, and it requires no code changes except for doing multi-device discharge using the device(n) clause. Due to the orthogonal design of the LLVM/OpenMP concurrent discharge mechanism, the plugin is also compatible with OpenMP asynchronous discharge (e.g., the nowait clause) [25].

In Figure 2, the client represents the OpenMP application, while the server is a binary provided by the remote offloading plugin. The user needs to run one instance of the client and one instance of the server per accelerator of the computing node. Once connected, the client can offload binary code to all the accelerators managed by all the servers. Both the host runtime and the device runtime plugin are working as they normally would, unaware of the client-server pair in the middle.

For each plugin interface call, the client serializes the function arguments and sends one blocking RPC request to the server. The server will deserialize incoming requests and execute the corresponding device operation, then reply to the client, so it can proceed. For a map(to/from) operation, the buffer must also be serialized. The server uses a task queue and a pool of threads to handle the RPC requests, so that offloading activities on one GPU does not block activities on other accelerators managed by the same server.

Originally, the remote offloading plugin uses gRPC [5] as its RPC backend, since it is a natural choice. Then an UCX [23] backend was added since it can utilize high-performance interconnects like Infiniband, thus bringing the potential to improve the offloading application's performance to a level comparable to that of its MPI+X equivalent. Based on the experimental results of [15], the UCX version of the plugin has obvious advantages compared to the gRPC version of the plugin, therefore, **we only consider the UCX-based version when discussing older versions of the plugin**.

2.3 Related Work

2.3.1 Remote GPU offloading. After the birth of OpenMP target offloading, [7] uses LLVM OpenMP target offloading to run on multi-node devices. However, the results show a significant performance disadvantage of the plugin compared to the MPI version of the same benchmark, HMMER. For the later implementation, OmpCloud [28] uses Apache Spark nodes for communication, fault tolerance, and load balancing management. This work treats the entire Spark cluster as a standalone OpenMP target offloading device. OmpCluster [29] makes fuller use of the computational performance of each node, but its plugin does not support computation using accelerated devices e.g. GPUs via pure OpenMP code. If a developer wants to perform computation with OmpCluster and accelerated devices, then the developer must implement device acceleration through vendor-specific APIs such as CUDA, HIP or OpenCL.

Besides modifying the OpenMP target offloading plugins, replacing MPI+X with a task-based programming model also received much attention, especially replacing the MPI+CUDA combination. Among them, Charm++ [1], UPC++ [2], Legion [3], and Chapel [4] NVSHMEM [6] enables remote communication directly from the CUDA kernels. Intel Cluster OpenMP uses a distributed shared memory runtime system to run OpenMP CPU parallel regions across nodes [24]. Kokkos Remote Spaces [8] is an extension to the Kokkos programming model [27] to support distributed shared memory for programming GPUs and other devices. rCUDA [20] is a framework for remote GPU virtualization, in which a set of GPUs can be shared and remotely accessed by several clients simultaneously.

2.3.2 Minimod. Minimod [11] is the industry application used in this study. It is developed and maintained by TotalEnergies E&P Research and Technology US. Implementations of Minimod have been evaluated with OpenMP tasks [19], and in distributed setups using the Legion programming model targeting CPUs [18] and GPUs [16, 22]. The present paper evaluates a version of Minimod using OpenMP target regions wrapped in tasks to make use of multiple GPUs simultaneously. To utilize the specification of locality-aware offloading in the new MPI-based version, we modified Minimod accordingly. The corresponding modifications will be introduced in subsection 5.3.

3 IMPLEMENTATION

The MPI-based version of the plugin is an LLVM/OpenMP target plugin. Compared to the UCX-based version of the plugin, it is compiled to contain only a dynamic library file named libomptarget.rtl.mpiremote.so and does not have a dedicated server. Figure 3 shows the MPI-based version of the workflow.

As a plugin for OpenMP target, the MPI-based offloading plugin controls the program's running by running a newly defined `Manager` class. This class will be responsible for the entire operation of the plugin, and the operation of MPI with the corresponding interface is also done through this class.

During the initialization phase of the program, the `Manager` collects the local device target offloading table by calling the load_binary() function and then broadcasts it to other nodes through MPI communication world. With the information collected from all other

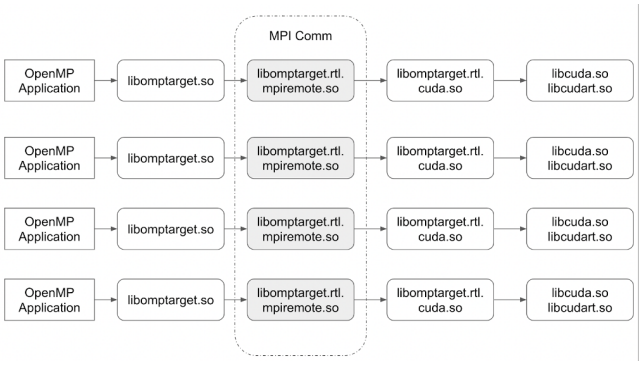

Figure 3: MPI-based LLVM/OpenMP remote offloading workflow for CUDA devices

nodes, the `Manager` will build a complete target offloading table, including all devices for subsequent use. When the program needs to distribute data, the `Manager` calls the corresponding MPI interfaces such as MPI_Send() and MPI_Recv(). When the program needs to execute a specific target code region, the `Manager` then executes the dynamic library of the corresponding underlying plugin. Limited by the design pattern of the OpenMP target, we did not break the original Host-Device model. Therefore, we set Node 0 as the host node and the other nodes as device nodes. Therefore, for the dataSubmit() function and dataRetrieve() function, all data will be sent out or received by Node 0's CPU memory.

Among the many gas pedals supported by OpenMP target offloading accelerators, NVIDIA GPUs are the ones receiving the most attention. Therefore, to further reduce the overhead when using NVIDIA GPUs, we utilize NVIDIA GPUDirect technology. GPUDirect has been helpful in two ways for this work. It could improve data transmission performance in both intra-node and inter-node scenarios. The intra-node data transfer mainly refers to GPUDirect Peer to Peer (P2P). GPUDirect P2P refers to GPU-GPU data copying and loading directly through the memory fabric (e.g. PCIe and NVLink) and not through the node memory. The inter-node data transfer mainly refers to GPUDirect RDMA technology. Remote direct memory access (RDMA) enables peripheral PCIe devices direct access to GPU memory. Designed specifically for the needs of GPU acceleration, GPUDirect RDMA provides direct communication between NVIDIA GPUs in remote systems. This eliminates the need for a buffered copy of data between the system CPU and through system memory, and according to NVIDIA [12], this technology can increase data transfer performance by 10 times.

4 IMPROVEMENT ON MPI-BASED VERSION

In this Section, we will focus on the improvements of the MPI-based version compared to the UCX-based version. We will first introduce the architectural changes due to the change of the underlying communication protocol, then we will introduce the concept of locality-aware offloading, and finally, we will introduce the improvement for ease-of-use.

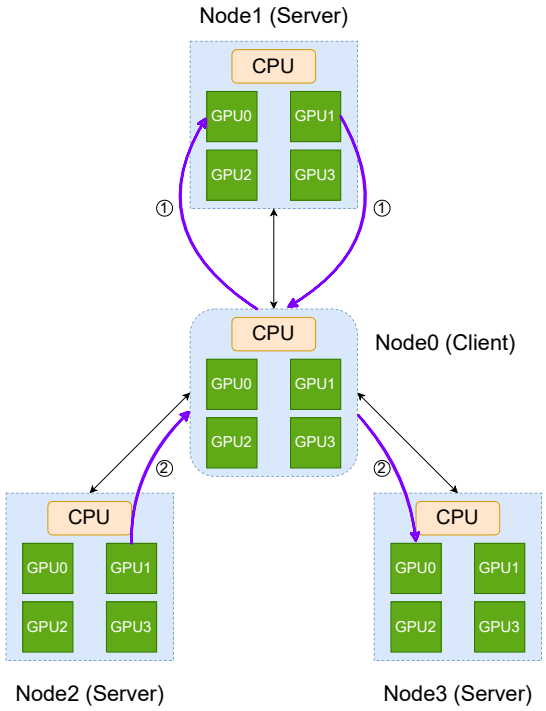

Figure 4: Centralized model in UCX-based plugin for CUDA devices

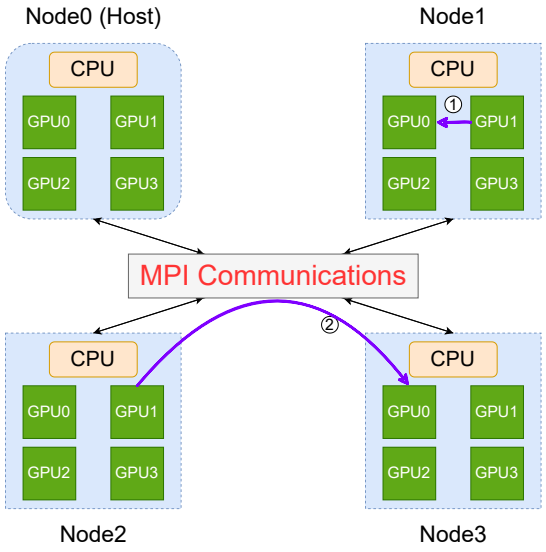

Figure 5: Decentralized model in MPI-based plugin for CUDA devices

4.1 "Decentralized" Model

In the UCX-based version, as shown in Figure 4, the client node is the control center for all nodes and devices, and all data storage and transmission instructions go through the client, the central node. On the one hand, due to the limited number of cores in the central node, a large number of instructions are difficult to be processed in time, causing an instruction transmission bottleneck; on the other hand, since all data will pass through the central node, the central node will have a memory shortage problem.

In the MPI-based version, as shown in Figure 5, the architecture becomes "decentralized". Due to the nature of MPI itself, we no longer need a so-called central node as the instruction center to send the corresponding instructions to each node, and the instruction communication bottleneck is solved.

At the same time, MPI enables the exchange of data between any two nodes, which provides the basis for different data exchange methods between Intra-node and Inter-node. The ① and ② in Figure 4 and Figure 5 show two scenarios separately.

The ① shown is sending data from GPU1 of Node1 to GPU0 of Node1. Both GPUs are located in the same node. In the UCX-based version, due to the framework of the plugin, the two GPUs cannot communicate with each other, which means that even if they are located in the same node, Node1's GPU1 needs to send data back to the client node first, and then the client node will send data back to Node1's GPU0. This means that the same data needs to be transmitted twice in the network, introducing network overhead twice. At the same time, since the data needs to be temporarily stored in the CPU memory of the central node before being received by GPU0 of Node1, this will undoubtedly increase the unnecessary memory overhead. However, in the MPI version, due to our redesigned plugin architecture, two devices located in the same node can communicate with each other, and with the GPUDirect function enabled, they can exchange data directly through GPUDirect P2P, which not only saves network overhead but also saves the memory usage of Host node and Node1 itself. In devices that do not support GPUDirect, the data will be transferred through GPU device to device (D2D) or other interfaces. Since this process does not need to go through the network transmission, its advantage in terms of network overhead optimization remains.

The ② shows that data is sent from GPU1 of Node2 to GPU0 of Node3. In the UCX-based version, which is also limited by the framework design of the plugin, the server nodes cannot communicate with each other. GPU1 of Node2 needs to send data back to the client node first, and then client node sends back to GPU0 of Node3. The disadvantage of this is that since the data transfer needs to go through client node, there is two RPCs' overhead in one transmission. Likewise, this also occupies the memory of the client node. In the MPI-based version, different nodes can communicate directly with each other. With GPUDirect enabled, the data from GPU1 of Node2 can be directly transferred to the network card via the PCIe interface, then transferred from InfiniBand to the network card of Node3, and transferred to GPU0 of Node3 via the PCIe interface, without taking up the memory of the node and with only one RPC overhead.

4.2 Locality-aware offloading

To better handle hierarchical computing and multi-node devices in heterogeneous computing and thus further improve performance, we propose the concept of locality-aware offloading in the MPI-based OpenMP offloading plugin. Locality-aware offloading refers

to using different policies for data transmission according to the hierarchy of devices, which means that network communication performance can be better utilized, thus reducing unnecessary network overhead. Meanwhile, for the side of OpenMP developers, they needn't know the practical devices hierarchy.

For example, we refactor function omp_target_memcpy() to use different methods for different data exchanges. When the devices (GPUs) are located in the same node, we exchange data inside the node by calling the P2P mentioned in section 3. And only when the devices are located in different nodes, we call the MPI-related communication functions for data exchange between nodes. For developers, there is no need to consider whether it is intra-node communication or inter-node communication; the plugin will automatically choose a more reasonable strategy when it runs.

In the future, we hope to extend this concept to hierarchical offloading, which could help developers further leverage the hierarchy of heterogeneous computing devices. For example, when a host node needs to distribute data to other compute nodes, there is often a large amount of duplicate data being sent. Hierarchical offloading can have the host node send only one copy of the data to the memory of the compute node, which is then replicated by the compute node and distributed to the individual accelerator devices on its node. In this case, the network overhead of transmitting data will be further reduced.

4.3 Ease-of-use

Remote OpenMP Offloading was initially designed to extend the availability of pure OpenMP code to support multi-node accelerated device offloading computations. Compared to MPI+X and CUDA coding, remote OpenMP offloading can significantly reduce the learning curve for developers of HPC programs, making their applications more effective when using accelerators with less development effort. However, the complexity of using the UCX-based version of the plugin runs counter to this original design intent.

On the one hand, restricted to the design of the UCX-based plugin, developers need to install the UCX and set up the corresponding software environment. If they want to turn on UCX-aware support, more steps are needed to configure it. When using UCX-based plugins, developers must manually set the server and client IPs and ports. For a typical multi-node application, the number of times you need to configure IPs is proportional to the number of nodes. This is cumbersome when using large accelerated-based HPC systems.

On the other hand, in UCX, the worker object provides independent progression and completion of communication operations[10]. This means that even if all traffic goes through the same network card, parallel injection from multiple workers can still improve message throughput through overlapping send-receive operations in the higher levels of the runtime system, especially when the combined message flow is not large enough to saturate the hardware bandwidth. Initially, the server uses a single worker to serve all the GPUs it has access. This means the worker must use locks to prevent data races caused by concurrent access from different threads, essentially serializing many overlap-able operations and reducing the injection rate [9]. In this way, if one wants to have the best performance, one have to use **one server per GPU** to reduce thread contention.

```
#Assign RPC method(gRPC or UCX)
os.environ['LIBOMPTARGET_RPC_TRANSPORT'] = 'UCX'

#Get IPs of nodes
proc = subprocess.run(['scontrol', 'show', 'hostnames',
     nodelist_short], capture_output=True, text=True)
nodelist = proc.stdout.splitlines()

#Setup IP and port on the client
rpc_address += convert_ip(nodelist[i])+':50050,'
#Repeat for device 1, 2, 3...

#Run client
if is_master:
    cmd_client = ['./XSBench','-m','event']
    proc = subprocess.run(cmd_client, capture_output=True
         , text=True)

#Run servers
if not is_master:
    my_env0 = os.environ.copy()
    my_env0['LIBOMPTARGET_RPC_ADDRESS'] = "0.0.0.0:50050"
    my_env0['CUDA_VISIBLE_DEVICES'] = "0"
    # Repeat on Device 1, 2, 3...
    cmd_server = ['openmp-offloading-server']
    subprocess.Popen(cmd_server,env=my_env0, shell=True)
    subprocess.Popen(cmd_server,env=my_env1, shell=True)
    subprocess.Popen(cmd_server,env=my_env2, shell=True)
    subprocess.run(cmd_server,env=my_env3, shell=True)
```

Figure 6: Python running script of XSBench on UCX-based plugin with slurm

```
srun --pty --exclusive -N 4 ./XSbench -m event
```

Figure 7: Running command of XSBench on MPI-based plugin with slurm

Figure 6 shows a typical Python script running UCX-based XSBench on Cypress (TotalEnergies's R&D HPC system, fully described in the next section) via slurm. As you can see, the developer needs to get the IP information of the node and the port information and start different programs on the client and server in turn. It is worth noting that the repetitive part of the script is greatly simplified in Figure 6, and the actual length of the script needed to run is much longer than the length shown. Figure 7 shows the slurm command to run MPI-based XSBench as a regular MPI program; using the MPI-based OpenMP offloading plugin does not require additional setup. If the developer is using a machine without slurm, such as the IBM Spectrum LSF used by ORNL's Summit, the MPI-based plugin can run as a regular MPI program, but the script to run the UCX-based plugin program may require very different run instructions, which creates a significant development effort.

5 EXPERIMENT

We evaluate our new MPI-based plugin using microbenchmarks and proxy-apps on the Cypress computing system at TotalEnergies R&T in Houston. Each Cypress node contains one AMD EPYC 7F52 16-core CPU, one Mellanox ConnectX-6 200 Gb/s Infiniband network card, and four NVIDIA A100 GPUs. The system runs CentOS 8 with Linux kernel 4.18.0, CUDA 11.5.119, and MOFED-5.1-2.5.8.0.

As the **Baseline**, we use commit `6120be4` of the original UCX-based remote offloading plugin, which is based on commit `67ab4c0` of the LLVM trunk. The remote offloading plugin with optimization in [10] is referred to as **Opt**. Since we can eliminate unnecessary overhead by allowing the client to offload to its local GPUs directly (instead of go through the remote plugin), we have included results that enabled client-side offloading, which we refer to as **Opt-L**. As our new MPI-based version, it is referred to as **New**.

5.1 Microbenchmarks

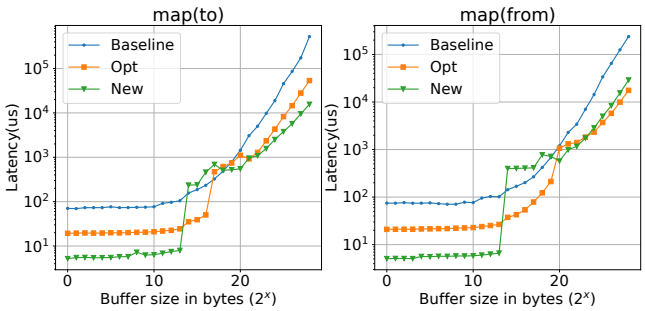

Figure 8: Remote GPU `map(to/from)` latency

Figure 8 shows the remote GPU *to/from* mapping latency of different buffer sizes. Compared to the baseline, the new MPI-based version of the plugin reduces the message latency by ~ 92% for small buffers and ~ 97% for large ones. Compared to the optimized UCX-based version(Opt), the new MPI-based version of the plugin reduces the message latency by ~ 73% for small buffers. For the large buffers, the *to* mapping and *from* mapping shows different results. For *to* mapping latency, the new MPI-based version reduces the message latency by ~ 70%; for *from* mapping latency, the new MPI-based version shows an acceptable enhancement.

The new MPI-based version shows little to no speedup for buffer size around 2^{15} bytes. This is caused by the compound effects of UCX switching its internal communication protocol. A similar phenomenon happens in Opt with 2^{19} since MPI's underlying communication layer is also UCX. Because we used the same setting for all buffer sizes instead of the best configuration for each buffer size, especially with NVIDIA Direct enabled, this result is not optimal. For real applications, the user should adjust their configuration of UCX like UCX's protocol switching thresholds(`UCX_RNDV_THRESH`) and schemes(`UCX_RNDV_THRESH`), so that the latency of the most frequently mapped buffer sizes could be optimal.

5.2 XSBench

XSBench [26] is a proxy-apps that capture the core computation of the Monte Carlo neutron transport code OpenMC [21]. We use the OpenMP offloading version with the same modifications used in [15] to enable multi-device offloading. We run the proxy-apps on 1 to 16 GPUs. We verified the performance improvement of the new version of the plugin by testing the weak-scaling version and the strong-scaling version of XSBench, respectively. Kernels execution time, total host-device transfer size per kernel and the total number of host-device transfers per kernel are listed in Table 1.

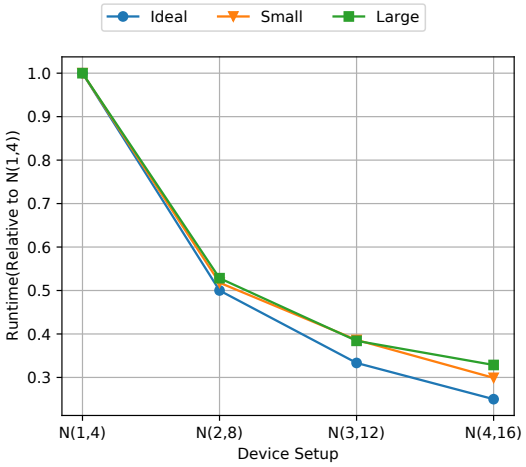

Figure 9: Strong-Scaling XSBench scaling

Strong scaling XSBench version results are presented in Figure 9. Based on [15] results, previous versions of the plugin scaled poorly when running the strong scaling XSBench, so they are not listed here. For the horizontal axis, $N(n, 4n)$ stands for running the benchmark on n nodes and using all $4n$ GPUs. It can be seen that the MPI-based version of the plugin shows good scalability both for large-size data and small-size data. The scalability of both is very close to that of the ideal behavior.

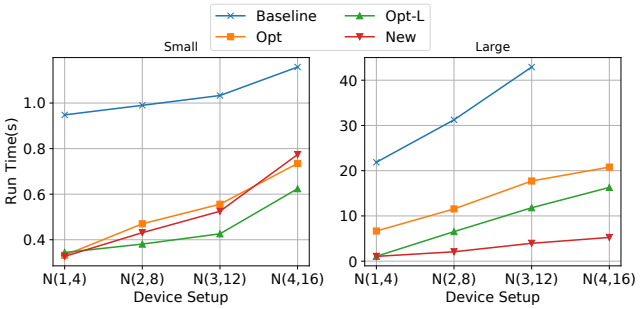

Figure 10: Weak-Scaling XSBench scaling

Weak scaling XSBench version results are presented in Figure 10. The baseline version plugin running the large data size crashes since it exhausts all available client host memory.

For the small data size, since the overhead percentage of transmitted data is very limited, the performance advantage of the MPI-based version cannot be highlighted at this time. There is no significant change compared to Opt and Opt-L, but there is still a big advantage compared to the baseline. For large-data size, the new version of the plugin can improve performance by up to ~ 68%.

5.3 Minimod

Minimod [11] is a proxy application that simulates the propagation of waves through subsurface models by solving a finite difference discretized form of the wave equation. In this work, we use one of

Table 1: XSBench kernel durations and per-kernel launch data transfers

	Kernel-Small	Transfer-Small	Kernel-Large	Transfer-Large	No. Transfers
XSBench	56.38 ms	240.4 MB	271.6 ms	5648 MB	19

```
for (int g = 0; g < nGPUs; g++) {
  #pragma omp task depend(...)
  #pragma omp target teams distribute parallel for device
      (g)
  for (...)
    // Stencil computation
}
// Halo exchange
for (int g = 0; g < nGPUs; g++) {
    // Left halo region: DtoH
    #pragma omp task depend(...)
    #pragma omp target update from(...) device(g)
    // Left halo region: HtoD
    #pragma omp task depend(...)
    #pragma omp target update to(...) device(g-1)
    // Repeat for the right halo regions ...
}
```

Figure 11: Simplified Minimod multi-GPU offloading and halo exchange workflow without locality-aware offloading

```
for (int g = 0; g < nGPUs; g++) {
  #pragma omp task depend(...)
  #pragma omp target teams distribute parallel for device
      (g)
  for (...)
    // Stencil computation
}
// Halo exchange
for (int g = 0; g < nGPUs; g++) {
    // Left halo region: DtoD or PtoP
    #pragma omp task depend(...)
    omp_target_memcpy(Ptr_Src, Ptr_Dst, DeviceID);
    // Repeat for the right halo regions ...
}
```

Figure 12: Simplified Minimod multi-GPU offloading and halo exchange workflow with locality-aware offloading

the kernels contained in Minimod: the acoustic isotropic propagator in a constant-density domain. This is the same kernel used in [18].

Minimod supports multi-device OpenMP offloading using `target` regions wrapped in OpenMP tasks (see Figure 11), and it shows good strong and weak-scaling behavior [19]. In this work, the 3D grid used in Minimod is partitioned along the X-axis (i.e. sliced parallel to the YZ-plane) regardless of the number of devices it is running on. Therefore, the amount of halo data exchanged between the devices is only related to the size of the cross-section of the grid, i.e., $dimY \times dimZ$. In the UCX-based version plugin, since the offloading servers are only connected to the client, not to each other, the halo data must all be relayed by the client, creating a central communication bottleneck. Minimod's kernel elapsed time and data transfer sizes for running on two devices are listed in Table 2.

In the new version of the MPI-based plugin, we can optimize the way of halo data exchange due to the support of locality-aware offloading. Figure Figure 12 shows the optimized code, and in comparison to the old version of the UCX-based plugin, the new version of the code contains fewer lines devoted to data exchange since the plugin runtime will choose (through **omp_target_memcpy()**) the specific way to manage data exchange through locality-aware offloading to achieve better performance.

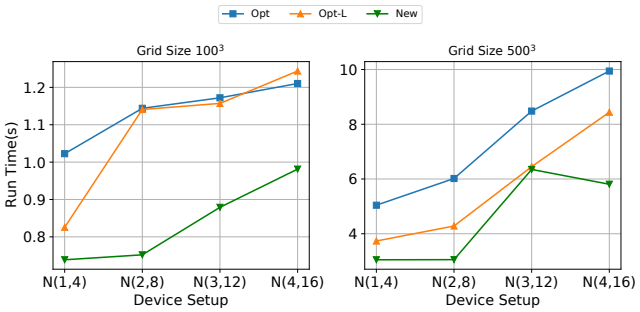

Figure 13: Results of Minimod with grid size 100^3 and 500^3

Minimod results with grid size 100^3 and 500^3 are presented in Figure 13. To better focus on the performance improvements of the new versions of the plugin, we discarded the subpar baseline results. The new MPI-based version outperforms Opt and Opt-L versions of the plugin significantly, showing the effectiveness of our improvement.

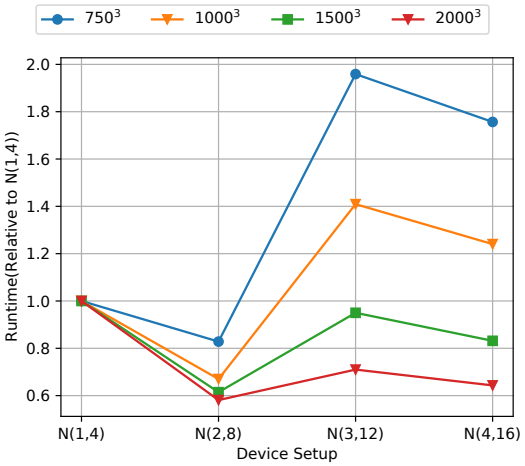

Figure 14: Minimod scaling results with new plugin on different large sizes

However, from strong scaling perspective, the results for Figure 13 are clearly not good enough (see [17]). It is well-known

Table 2: Minimod total kernel durations and data transfers (per iteration)

	Kernel-100^3	Transfer-100^3	Kernel-100^3	Transfer-500^3	No. Transfers
Minimod	171.9 μs	182.2 KB	6733 μs	4032 KB	2

(and reported in [10]), that as the number of devices increases, the execution time of the kernel decreases linearly, where the halo exchange overhead grows linearly, this is due to the domain decomposition of the problem. This problem is mitigated in the MPI version by replacing inter-node data exchange with intra-node data exchange, but it only marginally solved the problem. Therefore, we have further explored the problem of poor scalability exhibited by OpenMP-offloading-based Minimod.

First, in order to better characterize the problem, we obtained Figure 14 by increasing the size of the grid so that the amount of computation per iteration increases, thus reducing the proportion of the total time spent on data transfer overhead. According to Figure 14, the program execution time decreases with the number of devices in the range of 4-8 when the grid size increases to 750^3. However, when the number of devices increases further, the speedup decreases accordingly, and this happens as we continue enlarging the grid.

Then, we perform a simple theoretical analysis of the Minimod strong scaling behavior and obtained Table 3. When the number of devices is 1, the program crashes due to insufficient device memory. From Table 3, we can see that when the number of devices increases from 1 to 4, the computation of a single device decreases proportionally, but the overhead generated by data exchange does not increase significantly since intra-node data movement is quite efficient, and the overhead of inter-node communication is 0. This scenario changes when the number of devices increases from 4 to 8, the data exchange between nodes becomes a substantial overhead due to the emergence of inter-node communication. Nonetheless, since the computation of a single device also appears to be significantly reduced at this time, the program can still guarantee its strong scalability, when the computation is relatively large. Finally, when the number of devices further increases from 8 to 16, the computation time is minimal per single device, but the data exchange between nodes appears to be significantly increased, given the non-overlap between computation and communication. This phenomenon eventually leads to a sharp decline in the program's performance.

Table 4 shows the profiling results obtained by NVIDIA Nsight Systems[13] with a grid size of 1000^3. It can be seen that the profiling results are shown in Table 4 match precisely with the results of the theoretical analysis in Table 3.

Figure 15 and Figure 16 show screenshots from running the profiler for two-node and four-node Minimod, respectively. The blue rectangles in each "CUDA HW" row show task executions on the GPUs' kernel, and the red rectangles show the CUDA P2P. The gaps in the middle of the colored rectangle are occupied by data transfer between nodes over the network. While the two-node profiling shows a healthy GPU utilization, the four-node profiling results show the overall GPU utilization is not so high.

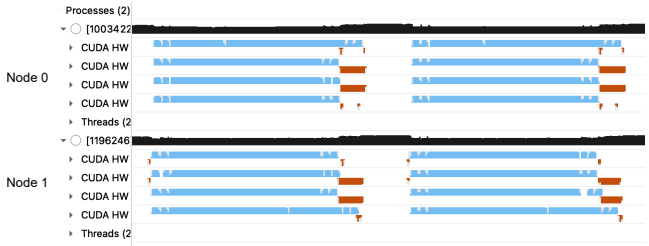

Figure 15: Sample NVIDIA Nsight Systems profiling time trace for two nodes

Figure 16: Sample NVIDIA Nsight Systems profiling time trace for four nodes

From the above analysis of Minimod, we do note that the MPI-based remote OpenMP offloading plugin overlaps communication and computation. Whereas, there is still room to improve the performance of the plugin.

6 CONCLUSION AND FUTURE WORK

MPI-based remote OpenMP offloading is an important step in the evolution of OpenMP. Its superior performance, ease of use, and better scalability and portability make it a good choice for distributed HPC developers. As a continuation of remote OpenMP offloading, the MPI-based plug-in can reduce the development burden of developers compared to MPI+X. Figure 17 shows the halo exchange part of Minimod MPI+CUDA version. Compared to the user-friendly API of remote OpenMP offloading, the code of the MPI+CUDA version is extremely cumbersome.

In this work, we use the refactored MPI-based version to address the performance bottlenecks presented in [10] and optimize them to achieve further performance improvements over the Baseline version. The evaluated performance using Microbenchmark shows a performance improvement of up to 70% over the previous version.

Table 3: Theoretical analysis of Minimod with grid size 2000^3

No. Nodes	No. Devices	Computation Per Device	Intra-Node Data Exchange Per Iteration	Inter-Node Data Exchange Per Iteration	Time For 2000^3
1	1	100%	0	0	Crash
1	4	25%	6	0	145.9850s
2	8	12.5%	12	2	84.8367s
4	16	6.25%	22	8	93.8965s

Table 4: Profiling results of Minimod with grid size 1000^3

No. Nodes	No. Devices	Kernel Time	Intra-Node Data Exchange Time (CUDA P2P Time)	Inter-Node Data Exchange Time (MPI Time)	Total Time
1	4	75.9s	2.348s	0s	19.9054s
2	8	38.7s	2.300s	18.234s	13.3376s
4	16	29.8s	2.887s	39.500s	24.6907s

In the XSBench evaluation, the latency was even reduced by 68% in the face of a weak-scaling version.

By replacing the underlying communication method, we have given more possibilities to remote OpenMP offloading - locality-aware offloading. This new feature is applied to the Minimod code, we have achieved a degree of "strong scalability" in offloading for the first time, which was not possible in the UCX-based versions [15] and [10] for large cases.

From Minimod's performance analysis and profiling, we can see that migrating a single-node OpenMP offloading program to a multi-node program based on the remote OpenMP offloading plugin will incur performance loss. This is because of two main reasons, on the one hand, increased inter-node overhead, and on the other hand, under-utilization of node-level computing resources.

The MPI-based version still has room for improvement. The plugin current version does not take advantage of alternative, and potentially more efficient runtime such as Partitioned Global Address Space (PGAS). The traditional two-sided communication between multiple nodes is still used, which is an inefficient mechanism for task synchronization. As next steps, we will optimize the synchronization method between nodes by enabling PGAS, to further reduce this overhead, one-sided communication will be exploited, this might be especially useful to handle some OpenMP target metadata (e.g. OpenMP target table).

For the insufficient utilization of resources hierarchy, we will further extend the concept of locality-aware offloading to hierarchical offloading. This will be achieved by giving developers new APIs, that map better to local computing resources configuration, that allow programs to achieve the highest possible performance on modern hierarchical heterogeneous computing devices.

In addition, as a plugin for OpenMP target, the MPI-based one only implements a minimal set of features. For example, asynchronous execution of the OpenMP target task is not yet supported. We will further improve the completeness of the remote OpenMP offloading function.

ACKNOWLEDGEMENTS

We would like to thank TotalEnergies E&P Research and Technologies US for their support of this work. This research was supported in part by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration.

REFERENCES

[1] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale. 2014. Parallel Programming with Migratable Objects: Charm++ in Practice. In *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 647–658. https://doi.org/10.1109/SC.2014.58

[2] John Bachan, Scott B. Baden, Steven Hofmeyr, Mathias Jacquelin, Amir Kamil, Dan Bonachea, Paul H. Hargrove, and Hadia Ahmed. 2019. UPC++: A High-Performance Communication Framework for Asynchronous Computation. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 963–973. https://doi.org/10.1109/IPDPS.2019.00104

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. 2012. Legion: Expressing locality and independence with logical regions. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1109/SC.2012.71

[4] B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442 arXiv:https://doi.org/10.1177/1094342007078442

[5] gRPC community. [n.d.]. gRPC. https://grpc.io/about/.

[6] Chung-Hsing Hsu, Neena Imam, Akhil Langer, Sreeram Potluri, and Chris J. Newburn. 2020. An Initial Assessment of NVSHMEM for High Performance Computing. In *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 1–10. https://doi.org/10.1109/IPDPSW50202.2020.00104

[7] Arpith C. Jacob, Ravi Nair, Alexandre E. Eichenberger, Samuel F. Antao, Carlo Bertolli, Tong Chen, Zehra Sura, Kevin O'Brien, and Michael Wong. 2015. Exploiting Fine- and Coarse-Grained Parallelism Using a Directive Based Approach. In *OpenMP: Heterogenous Execution and Data Movements*, Christian Terboven, Bronis R. de Supinski, Pablo Reble, Barbara M. Chapman, and Matthias S. Müller (Eds.). Springer International Publishing, Cham, 30–41.

[8] Kokkos. [n.d.]. Kokkos Remote Spaces. https://github.com/kokkos/kokkos-remote-spaces

[9] Wenbin Lu, Tony Curtis, and Barbara Chapman. 2019. Enabling Low-Overhead Communication in Multi-threaded OpenSHMEM Applications using Contexts. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*. 47–57. https://doi.org/10.1109/PAW-ATM49560.2019.00010

[10] Wenbin Lu, Baodi Shan, Eric Raut, Jie Meng, Mauricio Araya-Polo, Johannes Doerfert, Abid M. Malik, and Barbara Chapman. 2022. Towards Efficient Remote OpenMP Offloading. In *OpenMP in a Modern World: From Multi-device Support to Meta Programming*, Michael Klemm, Bronis R. de Supinski, Jannis Klinkenberg, and Brandon Neth (Eds.). Springer International Publishing, Cham, 17–31.

```
// Make sure the MPI is initialized and that we have
     sufficient thread support
if(MPI_Initialized(&init) != MPI_SUCCESS || !init)
  return ERROR_INITIALIZATION_FAILED;
AsfResult r = checkMPIThreading();
if(r < SUCCESS) return r;

// Get some stats from the base communicator
if(MPI_Comm_size(commbase, &size) != MPI_SUCCESS) {
  mpi_exit();
  return ERROR_INITIALIZATION_FAILED;
}

// Figure out the dimensions we'll be working with.
if(dims[0] == 0 && dims[1] == 0 && dims[2] == 0) {
  // For simplicity we just let MPI do its thing. It
       might not be balanced but it'll work.
  if(MPI_Dims_create(size, 3, dims) != MPI_SUCCESS) {
    mpi_exit();
    return ERROR_INITIALIZATION_FAILED;
  }
} else if(dims[0] == 0 || dims[1] == 0 || dims[2] == 0) {
  mpi_exit();
  return ERROR_INITIALIZATION_FAILED;
}

// The allocations must be exact
if(dims[0] * dims[1] * dims[2] != size) {
  mpi_exit();
  return ERROR_INITIALIZATION_FAILED;
}

// Now that the sanity checks are done, allocate some
     space for us to work
AsfPCollective_MPI coll;
if(!(coll = malloc(sizeof *coll))) {
  mpi_exit();
  return ERROR_OUT_OF_MEMORY;
}

// Get our location in the entire grid
int rank;
if(MPI_Comm_rank(coll->communicator, &rank) !=
     MPI_SUCCESS) {
  MPI_Comm_free(&coll->communicator);
  free(coll);
  mpi_exit();
  return ERROR_INITIALIZATION_FAILED;
}
int coords[3];
if(MPI_Cart_coords(coll->communicator, rank, 3, coords)
     != MPI_SUCCESS) {
  MPI_Comm_free(&coll->communicator);
  free(coll);
  mpi_exit();
  return ERROR_INITIALIZATION_FAILED;
}

// Send message to everyone in the neighborhood and
     receive
// back the numbers of Cuboids to send to each.
for(size_t i = 0; i < cnt; i++) {
  if(MPI_Isend(...) != MPI_SUCCESS) {
    mpi_exit();
  }
}
for(size_t i = 0; i < cnt; i++) {
  if(MPI_Recv(...) != MPI_SUCCESS) {
    mpi_exit();
  }
}

//Corresponding CUDA code
for (uint64_t i = 0; i < width; ++i) {
  cudaMemcpy2D(...);
}
#if 0
  copyIn_DEV<<<dim3(width, height, depth),1>>>(...);
#endif
cudaDeviceSynchronize();
```

Figure 17: Simplified Minimod halo exchange with MPI+CUDA

[11] Jie Meng, Andreas Atle, Henri Calandra, and Mauricio Araya-Polo. 2020. Minimod: A Finite Difference solver for Seismic Modeling. *arXiv* (2020). arXiv:2007.06048 [cs.DC] https://arxiv.org/abs/2007.06048

[12] NVIDIA. [n.d.]. NVIDIA CUDA GPUDirect RDMA. https://docs.nvidia.com/cuda/gpudirect-rdma/index.html.

[13] NVIDIA. [n.d.]. NVIDIA Nsight Systems. https://developer.nvidia.com/nsight-systems.

[14] OpenMP Architecture Review Board. 2018. *OpenMP Application Programming Interface.* https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf Version 5.0.

[15] Atmn Patel and Johannes Doerfert. 2022. Remote OpenMP Offloading. In *High Performance Computing*, Ana-Lucia Varbanescu, Abhinav Bhatele, Piotr Luszczek, and Baboulin Marc (Eds.). Springer International Publishing, Cham, 315–333. https://doi.org/10.1007/978-3-031-07312-0_16

[16] Eric Raut, Jonathon Anderson, Mauricio Araya-Polo, and Jie Meng. 2021. Evaluation of Distributed Tasks in Stencil-based Application on GPUs. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2).* 45–52. https://doi.org/10.1109/ESPM254806.2021.00011

[17] Eric Raut, Jonathon Anderson, Mauricio Araya-Polo, and Jie Meng. 2021. Evaluation of Distributed Tasks in Stencil-based Application on GPUs. In *2021 IEEE/ACM 6th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2).* 45–52. https://doi.org/10.1109/ESPM254806.2021.00011

[18] Eric Raut, Jonathon Anderson, Mauricio Araya-Polo, and Jie Meng. 2021. Porting and Evaluation of a Distributed Task-Driven Stencil-Based Application. In *Proceedings of the 12th International Workshop on Programming Models and Applications for Multicores and Manycores* (Virtual Event, Republic of Korea) *(PMAM'21).* Association for Computing Machinery, New York, NY, USA, 21–30. https://doi.org/10.1145/3448290.3448559

[19] Eric Raut, Jie Meng, Mauricio Araya-Polo, and Barbara Chapman. 2020. Evaluating Performance of OpenMP Tasks in a Seismic Stencil Application. In *OpenMP: Portable Multi-Level Parallelism on Modern Systems*, Kent Milfeld, Bronis R. de Supinski, Lars Koesterke, and Jannis Klinkenberg (Eds.). Springer International Publishing, Cham, 67–81. https://doi.org/10.1007/978-3-030-58144-2_5

[20] Carlos Reaño, Federico Silla, Gilad Shainer, and Scot Schultz. 2015. Local and Remote GPUs Perform Similar with EDR 100G InfiniBand. In *Proceedings of the Industrial Track of the 16th International Middleware Conference* (Vancouver, BC, Canada) *(Middleware Industry '15).* Association for Computing Machinery, New York, NY, USA, Article 4, 7 pages. https://doi.org/10.1145/2830013.2830015

[21] Paul K. Romano and Benoit Forget. 2013. The OpenMC Monte Carlo particle transport code. *Annals of Nuclear Energy* 51 (2013), 274–281. https://doi.org/10.1016/j.anucene.2012.06.040

[22] Ryuichi Sai, John Mellor-Crummey, Xiaozhu Meng, Mauricio Araya-Polo, and Jie Meng. 2020. Accelerating High-Order Stencils on GPUs. arXiv:2009.04619 [cs.DC]

[23] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network APIs and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects.* 40–43. https://doi.org/10.1109/HOTI.2015.13

[24] Christian Terboven, Dieter An Mey, Dirk Schmidl, and Marcus Wagner. 2008. First Experiences with Intel Cluster OpenMP. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism* (West Lafayette, IN, USA) *(IWOMP'08).* Springer-Verlag, Berlin, Heidelberg, 48–59.

[25] Shilei Tian, Johannes Doerfert, and Barbara Chapman. 2022. Concurrent Execution of Deferred OpenMP Target Tasks with Hidden Helper Threads. In *Languages and Compilers for Parallel Computing*, Barbara Chapman and José Moreira (Eds.). Springer International Publishing, Cham, 41–56.

[26] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XS-Bench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future.* Kyoto. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[27] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. 2022. Kokkos 3: Programming Model Extensions for the Exascale Era. *IEEE Transactions on Parallel and Distributed Systems* 33, 4 (2022), 805–817. https://doi.org/10.1109/TPDS.2021.3097283

[28] Hervé Yviquel, Lauro Cruz, and Guido Araujo. 2018. Cluster Programming Using the OpenMP Accelerator Model. *ACM Trans. Archit. Code Optim.* 15, 3, Article 35 (aug 2018), 23 pages. https://doi.org/10.1145/3226112

[29] Hervé Yviquel, Marcio Pereira, Emílio Francesquini, Guilherme Valarini, Pedro Rosso Gustavo Leite, Rodrigo Ceccato, Carla Cusihualpa, Vitoria Dias, Sandro Rigo, Alan Souza, and Guido Araujo. 2022. The OpenMP Cluster Programming Model. *51st International Conference on Parallel Processing Workshop Proceedings (ICPP Workshops 22)* (2022).