IMPACC: A Tightly Integrated MPI+OpenACC Framework Exploiting Shared Memory Parallelism

Jungwon Kim
Oak Ridge National Laboratory
kimj@ornl.gov

Seyong Lee
Oak Ridge National Laboratory
lees2@ornl.gov

Jeffrey S. Vetter
Oak Ridge National Laboratory
vetter@ornl.gov

ABSTRACT

We propose IMPACC, an MPI+OpenACC framework for heterogeneous accelerator clusters. IMPACC tightly integrates MPI and OpenACC, while exploiting the shared memory parallelism in the target system. IMPACC dynamically adapts the input MPI+OpenACC applications on the target heterogeneous accelerator clusters to fully exploit target system-specific features. IMPACC provides the programmers with the unified virtual address space, automatic NUMA-friendly task-device mapping, efficient integrated communication routines, seamless streamlining of asynchronous executions, and transparent memory sharing. We have implemented IMPACC and evaluated its performance using three heterogeneous accelerator systems, including Titan supercomputer. Results show that IMPACC can achieve easier programming, higher performance, and better scalability than the current MPI+OpenACC model.

CCS Concepts

•Software and its engineering → Distributed programming languages; Concurrent programming languages; Source code generation; Runtime environments;

Keywords

MPI; OpenACC; Clusters; Heterogeneous computing; Programming models

1. INTRODUCTION

High Performance Computing (HPC) systems are becoming deeply hierarchical and heterogeneous. Instead of expensive shared memory systems, distributed memory systems, such as clusters that consist of low cost commodity servers, have become the mainstream in HPC architectures [9]. A typical cluster node is equipped with multi-socket, multi-core CPUs in a shared NUMA memory architecture. The addition of accelerators, such as NVIDIA/AMD GPUs, MICs (Intel Xeon Phi coprocessors), and FPGAs further increase the levels of the hierarchy and heterogeneity[48].

During this move to clusters in HPC environments, a distributed memory message passing model has played a pivotal role. The Message Passing Interface (MPI) is a message passing API, and it has been widely accepted as the de facto industry standard. This enables the HPC programmer to develop portable and scalable parallel applications using MPI, which can then execute on nearly all HPC systems.

Unlike MPI, there was no industry standard for accelerator programming models. Early on, each accelerator had its own programming language or methodology: CUDA for NVIDIA GPUs, Brook+ for AMD GPUs, LEO for MICs, and HDL for FPGAs. In order to achieve code portability, the Khronos Group introduced OpenCL as a standard for accelerator programming model. However, OpenCL does not provide ease of programming and performance portability because it exposes low-level hardware architecture to the programmers [41, 45].

Later, the OpenACC API [3] was designed to provide an easy, high-level, and portable way to develop parallel programs across a wide range of accelerators. OpenACC simplifies the accelerator programming through the use of directives, like the OpenMP API. Based on directives supplied by the programmers, OpenACC compilers automatically offload compute-intensive code to an accelerator, managing data movement between the host CPU and the accelerator. However, OpenACC and most of existing accelerator programming models are intra-node programming models, assuming node-scale systems (*host + accelerator*), where one or small number of accelerators are attached to the host CPU. This design, in turn, limits scalability [37].

In order to write portable and scalable applications for heterogeneous accelerator clusters, a hybrid *MPI+OpenACC* programming model can be feasible. It inherits the advantages, such as high performance, scalability, and portability from MPI and programmability and portability from OpenACC. However, this mixture of two orthogonal programming models introduces some inefficiencies and complexities including redundant data movement and excessive synchronization between the models.

In addition, a deeper hybrid programming model with MPI across nodes, shared memory parallel programming models, such as OpenMP within a node, and OpenACC for accelerators has some advantages over the flat MPI+OpenACC hybrid model[20, 35, 51]. It can exploit a shared memory parallelism in the system, such as data sharing, efficient buffer management without inter-process

HPDC'16, May 31-June 04, 2016, Kyoto, Japan

 ISBN 978-1-4503-4314-5/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2907294.2907302

communication, and direct data transfer between accelerators on the same node[6, 43]. However, the deeper hybrid programming model exposes deeper memory and architecture hierarchy to the programmers, which is more complex and harder to program, resulting in lower productivity.

1.1 Contributions

In this paper, we propose *IMPACC* (*I*ntegrated *M*essage *P*assing *ACC*elerator programming), a novel programming framework for heterogeneous accelerator clusters; IMPACC tightly integrates MPI and OpenACC, while exploiting the shared memory parallelism in the system. IMPACC dynamically adapts the input MPI+OpenACC application onto the target heterogeneous system to fully exploit target system-specific features. This enables IMPACC to provide programmers with easier programming, higher performance, and better scalability than the current MPI+OpenACC model in heterogeneous accelerator cluster programming. This paper makes the following contributions:

- IMPACC integrates task creation in MPI and accelerator assignment in OpenACC adaptively to the target system. Also, the IMPACC runtime pins each task to certain CPUs in a NUMA-friendly way, considering its assigned accelerator in order to avoid undesired communication overhead.
- The IMPACC runtime maps the device memories of all available accelerators in a node onto the single *unified node virtual address space*. IMPACC implements an MPI task as a lightweight user-level thread, and the threaded-MPI tasks on the same node run on the single unified node virtual address space.
- IMPACC presents *unified MPI communication routines* that integrate MPI communication routines and OpenACC accelerator memory copy operations, with the help of the unified node virtual address space. It eliminates redundant communication both between tasks and between MPI and OpenACC, and it leads to low communication overhead.
- IMPACC provides *unified activity queue* that integrates non-blocking MPI communication to the OpenACC asynchronous activity queue in order to allow seamless streamlining of asynchronous intra-node/internode communication. It eliminates synchronization between two orthogonal streamlines of MPI and OpenACC, resulting in better programmability and scalability.
- IMPACC proposes *node heap aliasing technique*. It enables the MPI tasks that are in the same node and that have a producer-consumer relationship to share data transparently to the application while keeping their MPI semantics the same.
- The programmers can exploit full features of IMPACC, such as unified MPI communication routines, unified activity queue, and intra-node data sharing by adding OpenACC directive extensions for IMPACC (*#pragma acc mpi*).
- We show the effectiveness of IMPACC by implementing the compiler and runtime. We evaluate its performance with three heterogeneous accelerator clusters (two NVIDIA GPU-based clusters (*PSG*[8] and *Titan*[1]) and one Intel MIC-based cluster (*Beacon*[18])). The evaluation results demonstrate that IMPACC achieves ease of programming, high performance, and scalability on extreme-scale systems.

2. IMPACC ARCHITECTURE OVERVIEW

This section describes the architecture of IMPACC using a hierarchy of models: platform model, programming model, execution model, and memory model.

2.1 Platform Model

The platform model for IMPACC is a heterogeneous accelerator cluster as shown in Figure 1. Heterogeneous accelerator clusters have been widely used in HPC because they offer opportunities to greatly increase computational performance within a limited power budget. Nominally, a cluster consists of one or more compute nodes. The nodes are connected by an interconnection network (e.g., InfiniBand, custom interconnect). Typically, the nodes share storage through a high performance parallel file system like Lustre and GPFS.

Each node is equipped with multi-socket, multi-core CPUs in a shared NUMA memory architecture and augmented with one or more accelerators (e.g., NVIDIA/AMD GPUs, MICs, FPGAs). An accelerator communicates with the CPUs and other accelerators via a peripheral interconnect, such as PCIe and NVLink. For GPUs, the accelerator consists of multiple fully parallel execution units. Each execution unit is multithreaded, and each thread on the execution unit supports SIMD or vector operations. Especially, in order to exploit all available computing resources fully, IMPACC considers a set of CPU cores as an accelerator.

2.2 Programming Model

IMPACC uses MPI+OpenACC as its programming model to provide programmers with a practical way to write scalable, portable, and high-efficient HPC applications for heterogeneous accelerator clusters.

MPI is a message passing library used for distributed memory systems. It is designed primarily to support Single Program, Multiple Data (SPMD) model. Its autonomous task execution, good locality, and fine-grained control enable MPI to achieve high performance and good scalability in large-scale systems. MPI is widely recognized as the de facto standard for parallel HPC programming in industry and academia, and most existing HPC applications are written in MPI [11, 49].

Rewriting large legacy HPC applications for heterogeneous accelerator clusters using CUDA, OpenCL, or HDL can be difficult, if not impractical. It requires significant rewriting and restructuring of hundreds to thousands of lines, and reorganization of major data structures.

To address this challenge, OpenACC was designed as a high-level accelerator programming model to provide code and performance portability across a wide range of accelerators. The OpenACC API contains a set of directive-based extensions to standard languages that enable offloading of compute-intensive code to accelerators. Because OpenACC is a directive-based model, it requires few modification of the legacy application code, and little restructuring of the code, if any. Moreover, the modified OpenACC application can be run as the unmodified original application by ignoring the OpenACC directives at compilation time.

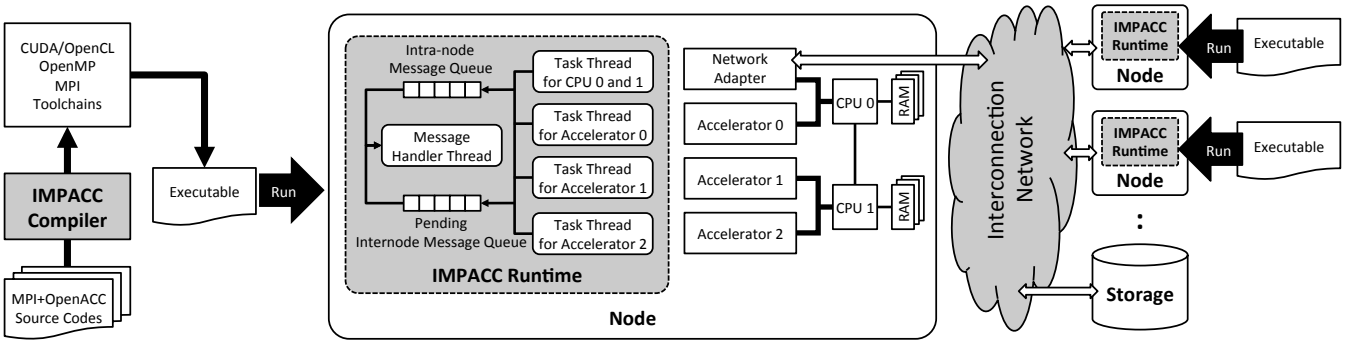

Figure 1: The Overview of the IMPACC Framework.

2.3 Execution Model

The IMPACC execution model exposes four levels of parallelism via *device*, *gang*, *worker*, and *vector* parallelism. Device parallelism follows the SPMD execution model. When IMPACC runs an MPI+OpenACC program on a heterogeneous accelerator cluster, the IMPACC runtime in every node launches the same number of program instances as the number of available accelerator devices in the node. The program instance is called an *MPI task*, or *task* for short. The IMPACC runtime implements a task as a lightweight user-level thread and assigns a distinct accelerator to each task as shown in Figure 1.

A task executes the host program in the input application on the host CPU with its attached accelerator device. Each task has its own unique id, called *rank*, throughout the whole running system. The tasks follow different execution paths to work on different data using the rank simultaneously in parallel. The device parallelism in IMPACC hides the hierarchy of nodes in the running cluster from the programmers, and thus it presents a flat-task view on the cluster.

A host program offloads compute intensive regions to the attached accelerator device. When a device executes a compute region, one or more gangs are launched on the device. Gang parallelism is coarse-grained. A gang has one or more workers. A worker is fine-grained and has a vector width. Vector parallelism is for SIMD or vector operations. Gang, worker, and vector parallelism are data parallelism. They are expressed as OpenACC *gang, worker, and vector* clauses, respectively.

2.4 Memory Model

The IMPACC memory model provides a single virtual address space, called *unified node virtual address space*, across the host system memory and device memories of all available accelerators in a node.

The IMPACC architecture assumes that the accelerator may be a discrete accelerator, such as an NVIDIA/AMD GPU, MIC and FPGA, or an integrated accelerator such as an AMD APU and a set of CPU cores. In the former case, the physical device memory on the accelerator is completely separate from the host memory. The IMPACC runtime dynamically maps their device memories in the unified node virtual address space. The host program performs all data movement between the host memory and device memory using OpenACC *data* constructs. A data construct allocates device memory and copies data between host and device memory. On the other hand, the integrated accelerator shares one system memory with the host. In this case, there is no need for additional memory mapping by the runtime and explicit memory management by the host program, and the operations can be elided.

In IMPACC, all MPI tasks in a node share the same unified node virtual address space because an MPI task runs as a user-level thread. The IMPACC runtime exploits the shared memory parallelism between tasks on the same unified node virtual address space to optimize the intra-node MPI communications.

The IMPACC architecture hides shared memory address space from the tasks for code portability. Instead, the IMPACC architecture employs the distributed memory model across the tasks both within a node and across nodes. All data is private to each task, and there is no coherence between tasks. A task performs computation using its local data and communicates with other tasks by calling MPI communication routines explicitly. This local view of IMPACC memory model naturally leads to good locality, which is essential to achieve good scalability on large-scale systems.

3. IMPLEMENTATION

Figure 1 illustrates the overview of the IMPACC framework. The framework consists of two parts, the IMPACC compiler and the IMPACC runtime.

3.1 The IMPACC Compiler

The IMPACC compiler is a source-to-source translator. The compiler translates the compute-intensive codes augmented by *parallel* or *kernels* construct in the input MPI+OpenACC source codes into accelerator kernel codes written in CUDA C and OpenCL C. It also generates the host program that orchestrates the execution of the kernels on accelerators from the input source codes. The compiler translates all global and static variables in the host program source code to thread-local variables to make them private to each threaded-MPI task. The details of the compiler implementation are out of the scope of this paper.

3.2 Automatic Task-Device Mapping

In the current MPI+OpenACC model, the user should provide the total number of MPI tasks when executing the application. In order to run the same number of tasks as the number of available accelerators in the cluster, the number of total tasks should be the number of nodes multiplied by the number of accelerators per node. Then each task sets its accelerator by calling OpenACC runtime routine

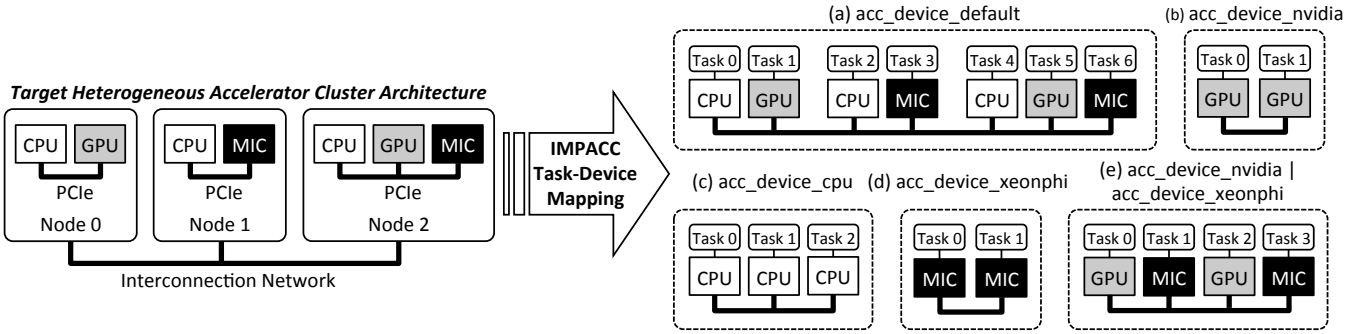

Figure 2: Automatic Task-Device Mapping in a Heterogeneous Accelerator Cluster with Different Nodes.

`acc_set_device_num()` with the local index of the accelerator. Typically the local index can be calculated by *MPI_rank % total_number_of_accelerators_per_node*.

This methodology assumes that the cluster consists of identical accelerator nodes; every node has the same number and same type of accelerators. When the target accelerator cluster consists of different accelerator nodes, the MPI task creation and OpenACC accelerator assignment are not simple.

In IMPACC, on the other hand, the user needs to provide the total number of nodes or the list of nodes when executing an MPI+OpenACC application. When the application is launched, the IMPACC runtime automatically creates the same number of MPI tasks as the number of all available or user's specified accelerators in the system. Users can specify the target accelerator types by setting an environment variable `IMPACC_ACC_DEVICE_TYPE`. Figure 2 (a), (b), (c), (d) and (e) shows the tasks created by the IMPACC runtime for the target heterogeneous accelerator cluster when the user specifies the target accelerator bit field type as `acc_device_default`, `acc_device_nvidia`, `acc_device_cpu`, `acc_device_xeonphi`, and `acc_device_nvidia | acc_device_xeonphi`, respectively.

The IMPACC runtime assigns a distinct accelerator to each task with a cluster-wide unique rank. Since the mapping between a task and an accelerator device is executed automatically by the IMPACC runtime, and thus the host program does not need to call `acc_set_device_num()`. The mapping is fixed during the application's lifetime, and the runtime ignores any additional `acc_set_device_num()` calls by the host program.

IMPACC does not provide any automatic mechanism for load balancing among different accelerators. Instead, the programmers can obtain the type of attached device to each task by calling a standard OpenACC runtime library routine, `acc_get_device_type()`. By using this device type information, the programmers can distribute the workload across the tasks manually.

3.3 NUMA-friendly Task-CPU Pinning

When multiple accelerator devices are equipped in a nonuniform memory access (NUMA) multi-CPU machine, it causes uneven data transfer performance between the CPUs and accelerators[39]. In Figure 1, the access distance between Accelerator 0 and CPU 0 is shorter than that of Accelerator 0 and CPU 1. The traffic in the former case traverses a PCIe between them. However, the latter case involves an additional traffic between CPU 0 and CPU 1 via an interconnect network such as Intel Quickpath Interconnect (QPI) and AMD HyperTransport (HT).

The IMPACC runtime identifies the CPU affinities of installed accelerators via *Linux sysfs (/sys/class/pci_bus)*. When the runtime creates the task threads for accelerators, it pins each task thread to the near CPU from its attached accelerator in order to avoid undesired communication overhead.

3.4 Unified Node Virtual Address Space

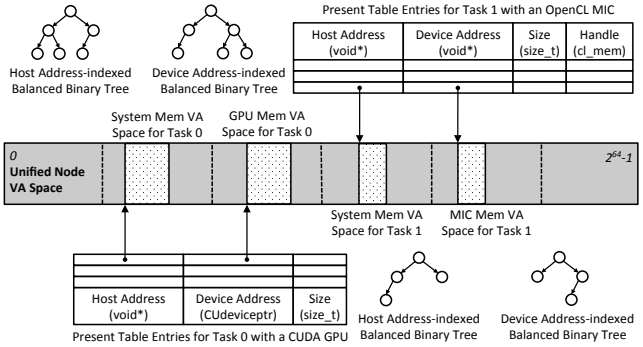

Figure 3: Present Tables for a CUDA GPU and an OpenCL MIC.

The *present table* in OpenACC maps host address ranges to corresponding device address ranges. The IMPACC runtime keeps a distinct present table for each task to avoid the access conflict between them. As shown in Figure 3, the runtime uses two balanced binary trees indexed by the host address and device address for a present table in order to reduce the worst-case search time.

A table entry in the present table stores the start address of host data, start address of the corresponding device data, and size of the data. When the runtime needs the device address associated with a host data, such as calling `acc_deviceptr()` from the host program, it searches the balanced tree indexed by the host address. The runtime searches the balanced tree indexed by the device address in the opposite case such as calling `acc_hostptr()`.

The IMPACC runtime maps the host system memory and the device memories of all available accelerators in a node to the single *unified node virtual address space*. A physical system memory address is translated to a virtual memory address, with the help of operating system and MMU. When the accelerator is a CUDA-enabled GPU, then the IMPACC

runtime uses the *Unified Virtual Addressing (UVA)* technique that CUDA provides. UVA maps the allocated GPU memory into the host's virtual address space. The address of the allocated GPU memory is expressed as `CUdeviceptr` as shown in the present table for Task 0 with a CUDA GPU in Figure 3.

In OpenCL, a memory object is a handle to a region of accelerator device memory. That is, the memory object does not indicate the address of accelerator device memory directly like `CUdeviceptr` in CUDA. To allocate a memory region in an OpenCL-enabled accelerator, the IMPACC runtime calls `clCreateBuffer()` and gets an OpenCL memory object handle expressed by `cl_mem`. Then, the runtime reserves a virtual memory address space by calling `malloc()` with the same size of the OpenCL memory object. The obtained virtual memory address space represents the device-side mapped address space for host data.

The table entry for OpenCL-enabled accelerator has separate fields with a device address in an address pointer and a memory handle in cl_mem shown in the present table for Task 1 with an OpenCL MIC in Figure 3. The runtime uses the mapped device memory address to look up the entry in the present table. Then, it calls underlying OpenCL runtime routines with the cl_mem in the found entry and a corresponding offset. Typically, `malloc()` is implemented as lazy allocation[29], and thus the reserved memory space does not consume the physical system memory.

3.5 Unified MPI Communication Routines

With the help of unified node virtual address space and the present table, the runtime can detect the data location from a virtual memory address. This enables unified MPI communication routines to transfer data between tasks. A task can send from or receive to the device memory of its attached accelerator by calling the MPI communication routines with a device-mapped address pointer.

There are two ways to transfer data in the device memory in the IMPACC environment. First, the task calls MPI routines using the device address pointer directly as follows:

```
1 MPI_Send(acc_deviceptr(host_data), cnt, type, dst, tag, comm);
```

This methodology modifies the original MPI code to use OpenACC runtime routines `acc_deviceptr()`, and it leads to poor portability. For better portability, IMPACC introduces a novel OpenACC directive extension for MPI as follows:

```
1 #pragma acc mpi sendbuf(device)
2 MPI_Send(host_data, cnt, type, dst, tag, comm);
```

The new OpenACC directive extension (`#pragma acc mpi`), called *IMPACC directive*, indicates that the immediately following MPI call uses the device memory of host data in the send buffer argument. It is augmented by a directive, and thus the compiler can ignore it to run the unmodified original application, resulting in better portability. The syntax of IMPACC directive is:

`#pragma acc mpi` *clause-list new-line*

where *clause* is one of the following:

`sendbuf(`*`[device] [,] [readonly]`*`)`
`recvbuf(`*`[device] [,] [readonly]`*`)`
`async` *`[(int-expr)]`*

The `sendbuf` and `recvbuf` clauses tell the compiler to use the device address of host data in the immediately following MPI call arguments and/or the buffers are read-only. When there is an `async` clause, the following non-blocking MPI call, such as `MPI_Isend()` and `MPI_Irecv()`, will be queued into an OpenACC asynchronous activity queue.

3.6 Unified Activity Queue

Figure 4 (a) shows a code snippet written in MPI+OpenACC. The program runs a kernel with `buf0` on the device and copies it to the system memory (line 2-3). Then the program sends `buf0` located in the system memory to another task (line 4) and receives `buf1` from the task (line 5). The program copies `buf1` from the system memory to the device memory and runs another kernel with `buf1` (line 6-7).

Figure 5 (a) shows the corresponding timeline graph for Figure 4 (a) code. The numbers in HOST-timelines in Figure 5 indicate that the code line numbers in Figure 4. The program runs six in-order operations; kernel - copyout - send - recv - copyin - kernel as shown in Figure 5.

In order to run the six operations sequentially, Figure 4 (a) uses synchronous APIs, such as OpenACC `kernels` construct, that has an implicit barrier at the end of the region, and blocking MPI communication routines. Synchronization operations are easy and safe to use, however it forces the host to wait until the completion of target operations as shown in Figure 5 (a). It leads to an undesirable waste of CPU cycles, resulting in poor CPU utilization. Furthermore, the synchronization can severely undermine scalability in the large-scale systems[44].

Figure 4 (b) shows an asynchronous version of Figure 4 (a). It uses MPI non-blocking communication routines and `async` clauses in `kernels` constructs. MPI non-blocking routines return immediately without waiting for the message to be sent or received.

In OpenACC, an accelerator has one or more *activity queues*. When there is an `async` clause on a `parallel`, `kernels`, `enter data`, `exit data`, or `update` directive, the host thread enqueues the parallel or kernels regions or data operations onto the device activity queue. The enqueued operations are processed independently and asynchronously on the device while the host thread continues its execution. The `async` clause may have a single nonnegative scalar integer expression argument. The argument is used to select the activity queue onto which to enqueue the operation on the accelerator.

With the help of MPI non-blocking communication and an OpenACC activity queue, Figure 4 (b) reduces the waste of CPU cycles as shown in Figure 5 (b). However, in order to synchronize two orthogonal streamlines across MPI and OpenACC, it still needs additional synchronization points such as `#pragma acc wait` and `MPI_Waitall()`. This leads to poor scalability.

IMPACC solves this synchronization issue by providing the *unified activity queue* to fully integrate MPI non-blocking communication and OpenACC activity queue. The users can enqueues MPI non-blocking communication routine calls to OpenACC activity queues. The IMPACC runtime completes the enqueued operations on an activity queue in order, while operations on different activity queues are active simultaneously and complete in any order.

```
/* (a) MPI+OpenACC Synchronous */
#pragma acc kernels loop copyout(buf0)
for (i = 0; i < n; i++) { buf0[i] = ...; }
MPI_Send(buf0, another_task);
MPI_Recv(buf1, another_task);
#pragma acc kernels loop copyin(buf1)
for (i = 0; i < n; i++) { ... = buf1[i]; }
```

```
/* (b) MPI+OpenACC Asynchronous */
#pragma acc kernels loop copyout(buf0) async(1)
for (i = 0; i < n; i++) { buf0[i] = ...; }
...
#pragma acc wait(1);
MPI_Isend(buf0, another_task, &req[0]);
MPI_Irecv(buf1, another_task, &req[1]);
...
MPI_Waitall(2, req);
#pragma acc kernels loop copyin(buf1) async(1)
for (i = 0; i < n; i++) { ... = buf1[i]; }
```

```
/* (c) IMPACC Unified Activity Queue */
#pragma acc kernels loop async(1)
for (i = 0; i < n; i++) { buf0[i] = ...; }
#pragma acc mpi sendbuf(device) async(1)
MPI_Isend(buf0, another_task, &req[0]);
#pragma acc mpi recvbuf(device) async(1)
MPI_Irecv(buf1, another_task, &req[1]);
#pragma acc kernels loop async(1)
for (i = 0; i < n; i++) { ... = buf1[i]; }
```

Figure 4: Synchronizations between MPI and OpenACC.

The `async` clauses in the IMPACC directive and `async` clauses in the `kernels` constructs in Figure 4 (c) have the same queue number `1`; these operations will be executed in-order by the runtime while the host thread is free from synchronization as shown in Figure 5 (c). It can improve the scalability of the system ultimately.

The unified activity queue also improves the programmability. In order to achieve the fully asynchronous communication, such as Figure 5 (c), in the current MPI+OpenACC model, it requires additional multithreaded programming to run a background thread that checks the status of non-blocking MPI requests or asynchronous OpenACC data transfers. This is much more difficult and error-prone than adding an `async` clause for the unified activity queue.

3.7 Handling MPI Communications

The IMPACC runtime runs a single *message handler thread* in each node as shown in Figure 1. The message handler thread handles all intra-node and pending fully asynchronous internode MPI communications. The message handler thread and task threads in each node share two in-order and lock-free multi-producer (task threads) single-consumer (message handler thread) queues, called *intra-node message queue* and *pending internode message queue*.

For the intra-node communication, when a task calls an intra-node MPI communication routine, the runtime creates a message command and enqueues the command into the intra-node message queue. The message handler thread matches a pair of a send message command and a receive message command in the intra-node message queue using their task ids and tags while guaranteeing their FIFO ordering. The message handler thread dequeues the matched pair of commands from the queue. The matched pair of message

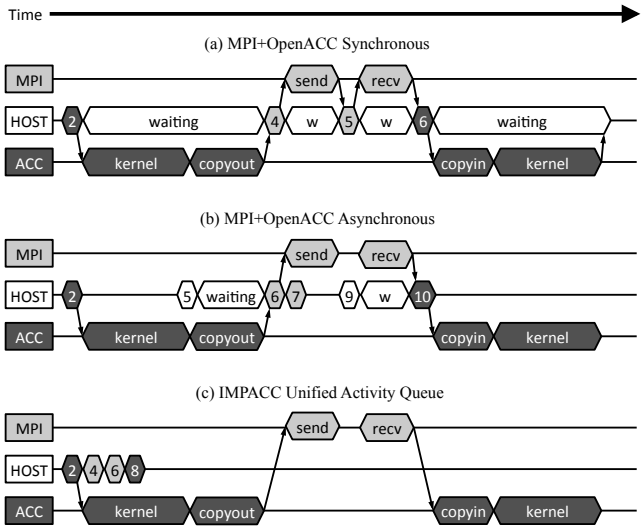

Figure 5: Synchronization Timeline.

commands represents a memory copy between host-to-host (HtoH), host-to-device (HtoD), device-to-host (DtoH), and device-to-device (DtoD) across the tasks depending on the locations of send and receive buffers.

The message handler thread fuses the matched two message commands into a single corresponding (accelerator) memory copy operation. We call it the *message fusion technique*. Figure 6 shows the differences between traditional MPI+OpenACC and IMPACC in the intra-node communication across tasks. In traditional MPI+OpenACC, a communication between tasks introduces inter-process communication and/or redundant host-to-host memory copy because a task is implemented as an OS process and runs on its private virtual address space.

On the other hand, the threaded-MPI tasks in IMPACC share a single unified node virtual address space, and the address of the send buffer and receive buffer in the matched MPI messages are located in the same linear virtual address space. Therefore, the runtime fuses two MPI messages into a single accelerator memory copy operation to eliminate the inter-process communication and redundant host-to-host memory copy as shown in Figure 6. Especially, when both send buffer and receive buffer reside in device memories, and the devices share the same upstream PCIe root complex, the runtime copies data directly between devices over the PCIe without the involvement of the CPU or system memory using the underlying vendor-specific features, such as NVIDIA GPUDirect[6] and AMD DirectGMA[4].

For the internode communication, the task threads call the underlying MPI communication routine in the system. When the data is located in the device memory, the runtime executes an additional memory copy between the device memory and the host system memory. If the underlying MPI library supports the direct accelerator memory access, such as Mellanox OFED GPUDirect RDMA[5], the runtime exploits it and transfers data directly from the device memory to a network adapter without staging through host memory.

The internode communication among the task threads depends on the multithreading support from the underlying MPI library. If the underlying MPI library supports

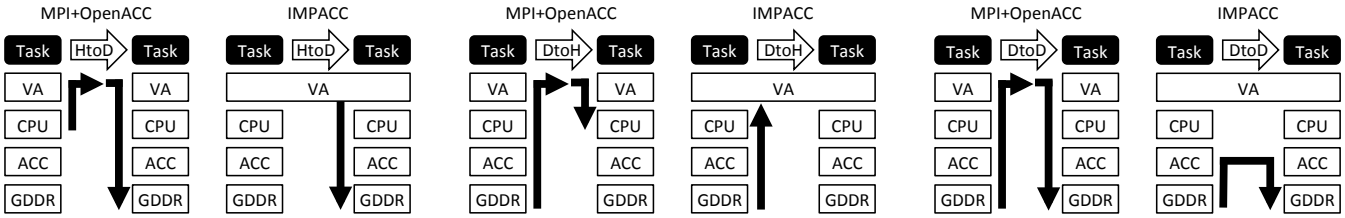

Figure 6: Message Fusion for Intra-node Communications. Two messages in MPI+OpenACC are fused into a single accelerator memory copy operation in IMPACC.

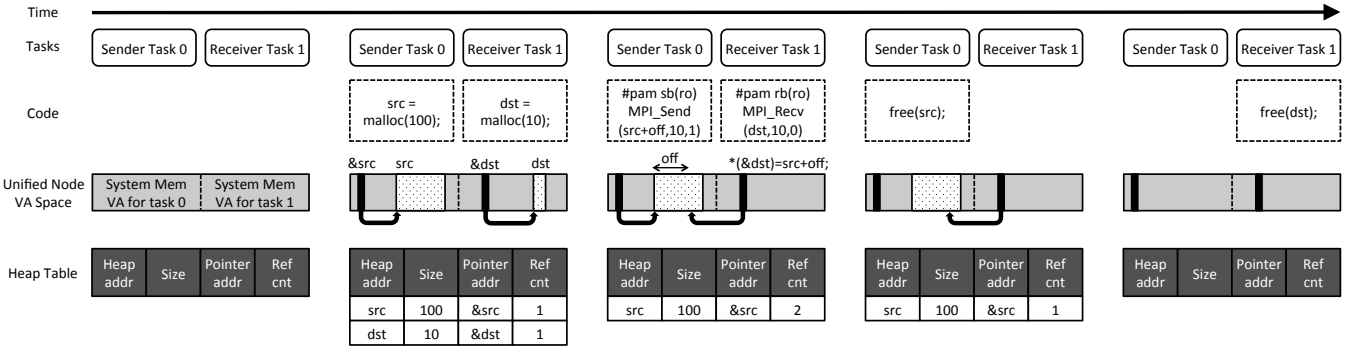

Figure 7: Node Heap Aliasing Example, where `#pam sb(ro)` and `#pam rb(ro)` refer to `#pragma acc mpi sendbuffer(readonly)` and `#pragma acc mpi recvbuffer(readonly)`, respectively.

`MPI_THREAD_MULTIPLE` level, then multiple task threads can call MPI routines, with no restrictions. Typical state-of-the-art MPI libraries that provide `MPI_THREAD_MULTIPLE` level use fine-grained locking support to decrease the contention between threads[1, 7, 16].

If the underlying MPI library does not provide multi-threading support, then the IMPACC runtime serializes the internode communication for each node to achieve thread safety. Therefore, as the number of MPI tasks per node increases, the serialization of communication may induce overhead, and result in poor scalability. However, IMPACC keeps a small number of MPI tasks per node as the same number of accelerators available. These alleviate the scalability issue caused by lock contention and serializing internode communication between task threads.

To implement fully asynchronous internode communication across MPI and OpenACC, the sender task thread reads the data from its device memory by calling an asynchronous device memory copy operation, such as `cuMemcpyAsync()` in CUDA or `clEnqueueReadBuffer()` with `CL_NON_BLOCKING` flag in OpenCL. Then the task thread sets a notification callback function to the memory copy operation, using `cuStreamAddCallback()` in CUDA or `clSetEventCallback()` in OpenCL. The callback function is executed when the memory copy operation completes. It calls `MPI_Isend()` to send the data to the target task in the different node asynchronously. For better performance, the runtime internally uses the pre-pinned host memory.

Meanwhile in the receiver task side, the receiver task thread receives the data from the sender task using `MPI_Irecv()`. Then it creates a new *pending internode message command* and adds it to a *pending internode message queue*. The pending internode message command contains the target address of the device memory and an `MPI_Request` object, MPI non-blocking communication request handle, to check its completion. The message handler thread checks the completions of pending commands in the internode pending message queue. When a pending command completes its non-blocking communication, the message handler thread calls `cuMemcpyAsync()` in CUDA or `clEnqueueWriteBuffer()` with `CL_NON_BLOCKING` flag in OpenCL to write data to the device memory and removes the pending command from the queue.

3.8 Intra-node Data Sharing

In the IMPACC framework, since the tasks on the same node share the unified node virtual address space, they can share data. The IMPACC directive has `readonly` attributes in the `sendbuf` and `recvbuf` clauses. There are five requirements to share data between tasks. First, the tasks should be on the same node. Second, the send buffer and receive buffer must be in the host heap memory. Third, every task calls MPI communication routines using the IMPACC directive with a `readonly` attribute. Fourth, the receiver task has no pointer variable that stores the receive buffer region before the MPI call. Last, the receive MPI call fully overwrites the receive buffer. When all these conditions are met, the IMPACC runtime enables the tasks to share the data between them using *node heap aliasing technique*. Node heap aliasing technique in IMPACC achieves data sharing among the MPI tasks that have a producer-consumer relationship transparently to the application while keeping their semantics the same.

Figure 7 shows an example. There are two tasks, Sender Task 0 and Receiver Task 1. Task 0 and Task 1 allocate heap `src` and `dst` with sizes of 100 and 10 respectively by

System	PSG	Beacon	Titan
Number of used (total) nodes	1 (16)	32 (48)	8,192 (18,688)
CPUs	2 × Intel Xeon E5-2698 v3	2 × Intel Xeon E5-2670	AMD Opteron 6274
Main memory size	256GB	256GB	32GB
Accelerators	8 × NVIDIA Kepler GK210	4 × Intel Xeon Phi coprocessor 5110P	NVIDIA Tesla K20x
Cores per accelerator	2,496 CUDA cores	60 x86 cores	2,688 CUDA cores
Accelerator core clock	875MHz	1.053GHz	732MHz
Memory per accelerator	12GB GDDR5	8GB GDDR5	6GB GDDR5
PCI Express	PCIe Gen3 x16	PCIe Gen2 x16	PCIe Gen2 x16
Interconnection	Mellanox InfiniBand FDR	Mellanox InfiniBand FDR	Cray Gemini interconnect
OS	CentOS 6.6	CentOS 6.2	Cray Linux Environment
Accelerator API	CUDA 6.5	Intel OpenCL 14.2	CUDA 6.5
MPI	MVAPICH2 2.0	Intel MPI Library 5.0	Cray MPICH2
MPI multithreading support	MPI_THREAD_MULTIPLE	MPI_THREAD_MULTIPLE	MPI_THREAD_MULTIPLE
C/C++ compiler	GCC 4.8.2	Intel C Compiler 15.0.2	Intel C Compiler 14.0.2

Table 1: The Target Heterogeneous Accelerator Systems.

calling `malloc()`. The IMPACC runtime hooks the heap-related routines, such as `malloc()`, `calloc()`, `realloc()`, `free()`, and etc., and it records the allocated heaps in the *Heap Table*. An entry in the Heap Table stores the allocated heap address, size, pointer address, and its reference count. Task 0 calls `MPI_Send()` augmented with an IMPACC directive with a readonly attribute (`#pragma acc mpi sendbuffer(readonly)`). Task 1 also calls `MPI_Recv()` with size 10 for the receive buffer augmented with an IMPACC directive with a readonly attribute (`#pragma acc mpi recvbuffer(readonly)`).

The message handler thread in the IMPACC runtime matches these two MPI communications and identifies that this intra-node communication meets all requirements for node heap aliasing. The message handler thread aliases the pointer of receive buffer (`dst`) to the send buffer address (`src + off`) and deallocates the original receive buffer heap region. Then, the message handler thread removes the corresponding heap table entry and increases the reference counter of the send buffer entry. After this sharing point, Task 1 shares the `src` heap region with Task 0 via `dst` pointer.

When a task calls `free()` to release the allocated heap region, the IMPACC runtime looks up the corresponding entry that contains the address of `free()` and decreases the reference count in the found entry. When the reference count becomes zero, it deallocates the heap region and removes the entry from the table.

MPI collective communications, such as `MPI_Bcast()`, in IMPACC also exploits node heap aliasing technique to eliminate redundant data movement. When the tasks call `MPI_Bcast()`, the IMPACC runtime sends the buffer in the root task to a task in every participated node. And then, the task that received the buffer sends the buffer to other tasks on the same node. At this time, if this data transfer meets the five requirements of the node heap aliasing technique, the tasks can share the buffer without additional memory copy.

4. EVALUATION

4.1 Methodology

Target systems. We evaluate the performance of IMPACC using three heterogeneous accelerator systems (*PSG*[8], *Beacon*[18], and *Titan*[1]). Table 1 summarizes the target systems.

Compiler and Runtime system. We have implemented the IMPACC compiler and runtime system. The IMPACC compiler is built on OpenARC[38], an open source OpenACC compiler framework. The accelerator part of the runtime is implemented with CUDA Driver API and OpenCL runtime.

Benchmark applications. We use four MPI+OpenACC applications: DGEMM, EP, Jacobi, and LULESH[33]. We compare their performance in our IMPACC framework with the legacy MPI+OpenACC implementations. In order to show the effectiveness of the runtime techniques proposed in the paper, we use the same IMPACC compiler for both cases to translate the OpenACC constructs to accelerator programs. The results show the performance gain from NUMA-friendly task-CPU pinning, intra-node/internode point-to-point and collective communication optimization, intra-node data sharing, and easier programming and better scalability from unified activity queue.

4.2 Results

NUMA-friendly Task-CPU Pinning. We measure the bandwidth of accelerator memory copies in order to show the effectiveness of NUMA-friendly task-CPU pinning in IMPACC. Bandwidth is measured using the transfers of blocks of data ranging in size from 64B to 1GB on the multi-socket CPUs systems: PSG and Beacon. A single task calls host-to-device (HtoD) or device-to-host (DtoH) memory copy with two configurations: NUMA-friendly and NUMA-unfriendly. We pin the task to the near CPU from the target accelerator in the NUMA-friendly configuration and pin the task to a far CPU in the NUMA-unfriendly configuration. Figure 8 demonstrates that our NUMA-friendly task-CPU pinning is effective for both HtoD and DtoH memory copies both on the NVIDIA GPU and Intel MIC systems. The NUMA-friendly configurations deliver higher bandwidth, up to 3.5 times, than NUMA-unfriendly configurations.

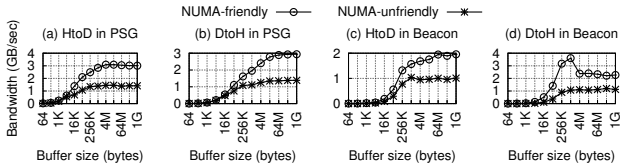

Figure 8: NUMA-friendly Task-CPU Pinning.

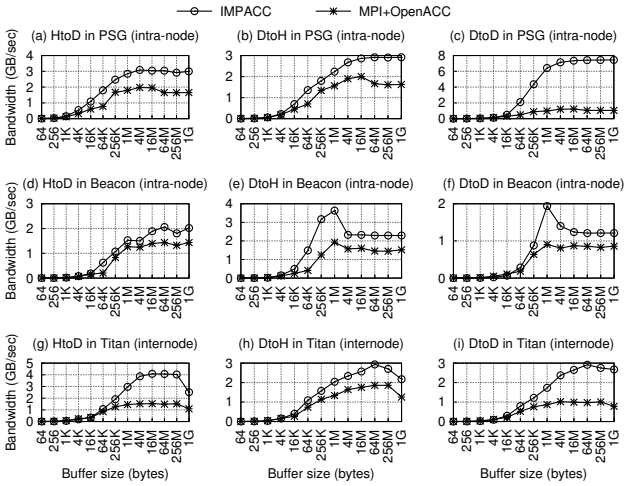

Figure 9: Point-to-point Communication Bandwidth.

Point-to-point Communication. Figure 9 shows the bandwidth of point-to-point communication between two tasks in IMPACC and MPI-OpenACC. We measure the bandwidth of intra-node point-to-point communication in PSG and Beacon and internode point-to-point communication in Titan. The IMPACC runtime eliminates the unnecessary host-to-host memory copy between the tasks on the same node as shown in Figure 6. IMPACC shows higher intra-node communication bandwidth than that of MPI+OpenACC as shown in Figure 9 (a) - (f). Especially, IMPACC shows almost eight times higher bandwidth than MPI+OpenACC in device-to-device intra-node communication in PSG (Figure 9 (c)) due to the direct transfer between devices via PCIe.

IMPACC also shows higher bandwidth in internode communications as shown in Figure 9 (g) - (i). This is because the IMPACC runtime internally exploits the Mellanox OFED GPUDirect RDMA from the underlying Cray MPICH2. It decreases the internode communication latency while eliminating the memory copy between the pinned CUDA buffer and the Mellanox HCA buffer.

DGEMM. DGEMM is a double-precision dense matrix-matrix multiplication, which computes the product of two matrices. We used square matrices for simplicity. The root task, whose rank is zero, sends the input sub-matrices to all of the other tasks, and then receives the output sub-matrices from them.

Figure 10 shows the strong scalability of DGEMM on the target systems. Figure 10 (a), (b), (c), and (d) show the speedup numbers of IMPACC and MPI+OpenACC over MPI+OpenACC with a single task in PSG. We vary the number of tasks (that is, number of devices) and the number of elements of the matrices from 1 to 8, and from 1K×1K to 8K×8K in powers of two, respectively. One DGEMM feature is that the ratio of computation ($O(N^3)$) to communication ($O(N^2)$) is proportional to $O(N)$. As shown in MPI+OpenACC implementation in Figure 10 (a), (b), (c), and (d), as the size of matrices increases, it shows better scalability because the large computation compensates the communication overhead.

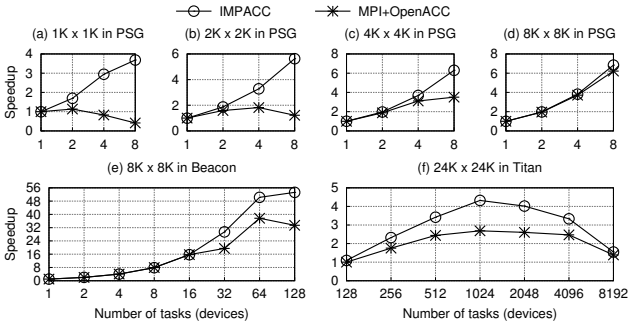

Figure 10: Speedup of DGEMM, normalized to MPI+OpenACC 1-task in PSG and Beacon, 128-tasks in Titan.

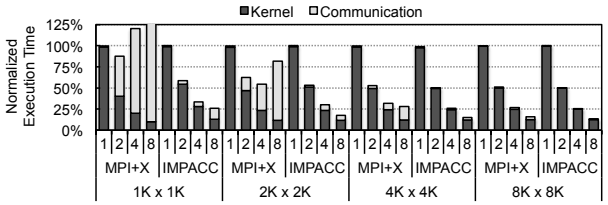

Figure 11: Execution Time Breakdown for DGEMM in PSG, normalized to MPI+OpenACC (noted as MPI+X) 1-task for each input. The x-axis represents the number of tasks (1, 2, 4, and 8) and the number of elements of the matrices.

On the other hand, IMPACC shows fair scalability for all input matrices even though MPI+OpenACC shows performance degradation with small-sized matrices. All tasks are on the same node, and the input matrices are read only. Therefore, IMPACC version exploits the node heap aliasing technique, and all the tasks share the root task's input matrices located in the unified node virtual address space. Also, higher point-to-point communication bandwidth in IMPACC decreases the communication overhead.

Figure 11 shows the normalized execution time breakdown for Figure 10 (a) - (d). The baseline is the total execution time of MPI+OpenACC version using a single task for each input. In the small input matrices, IMPACC dramatically reduces the communication overhead. As the size of matrices increases, the kernel execution time dominates the total execution time, and the communication overhead is hidden.

Figure 10 (e) shows the speedup numbers over the MPI+OpenACC version using a single task in Beacon. We vary the number of tasks from 1 to 128 in powers of two. A node contains 4 devices, and then 128 tasks run across 32 nodes. IMPACC shows similar performance to MPI+OpenACC until 16 tasks. However, IMPACC shows better performance than MPI+OpenACC from 32 tasks. As the number of tasks increases, the communication overhead increases because DGEMM broadcasts one of the input matrices to all tasks. IMPACC eliminates the redundant memory copy across the task on the same node in `MPI_Bcast()` using the node heap aliasing technique. Furthermore, IMPACC shows fair scalability up to 128 tasks even though MPI+OpenACC shows performance degradation in 128 tasks. This is because IMPACC version elimi-

nates all synchronization points, which undermine scalability, in the host program using the IMPACC unified activity queues. It only needs to add IMPACC directives along with `async` clauses.

Figure 10 (f) shows the speedup numbers over the MPI+OpenACC version using 128 tasks in Titan. We set the number of elements in the matrices to the maximum (24K×24K) that 128 tasks can hold their sub-matrices in their device memories. Both IMPACC and MPI+OpenACC show performance degradation from 1024 nodes due to its communication overhead. However, IMPACC shows better performance than MPI+OpenACC up to 160% in 1024 nodes. This performance gain comes from exploiting the unified activity queues and higher internode communication bandwidth in IMPACC as shown in Figure 9 (g) - (i).

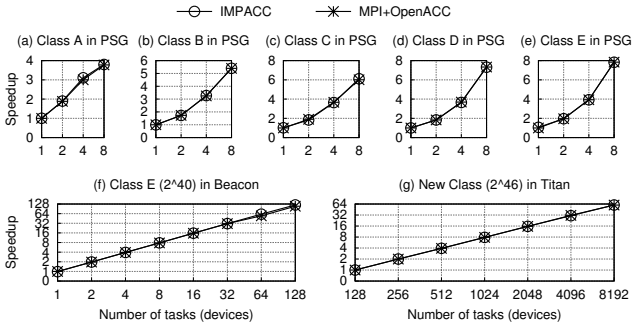

Figure 12: Speedup of EP, normalized to MPI+OpenACC 1-task in PSG and Beacon, 128-tasks in Titan.

EP. EP is Embarrassingly Parallel kernel in NAS parallel benchmarks[15]. It requires no communication between tasks except for the final reduction, and the kernel execution time dominates the total execution time.

We vary the input size from class A to class E in PSG as shown in Figure 12. In class D and E, EP shows almost linear scalability. However, in class A, B, and C, as the input size decreases it shows poor strong scalability. This is because, as the number of tasks increases, the kernel workloads to each task are reduced, and it lowers the device utilization.

Figure 12 (f) shows the speedup of class E on Beacon, and it shows a linear scalability. Figure 12 (g) shows the speedup of our new class, which is 64 times bigger than the NPB's biggest class (class E), over 128 tasks. Both IMPACC and MPI+OpenACC show linear speedups.

EP shows almost same performances in IMPACC and MPI+OpenACC for all experiments. IMPACC has no room to optimize performance for EP that has very little communication overhead.

Jacobi. The Jacobi iteration method is a stencil computation used to solve partial differential equations. We evaluate 2D Jacobi iteration for a square matrix and partition the matrix in one dimension for the given tasks. Communications between tasks are needed at block boundaries in order to receive values of neighbor points that are owned by another task.

Figure 13 (a), (b), (c), and (d) show speedup numbers of IMPACC and MPI+OpenACC over the MPI+OpenACC version with a single task in PSG. We vary the number of elements in mesh from 1K×1K to 8K×8K. IMPACC shows better performance than MPI+OpenACC due to its optimized intra-node communication using direct device-to-device memory copy.

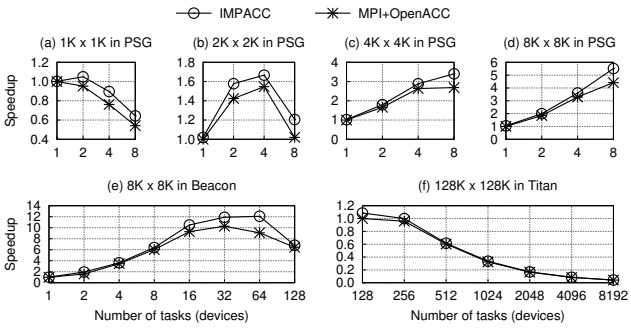

Figure 13: Speedup of Jacobi, normalized to MPI+OpenACC 1-task in PSG and Beacon, 128-tasks in Titan.

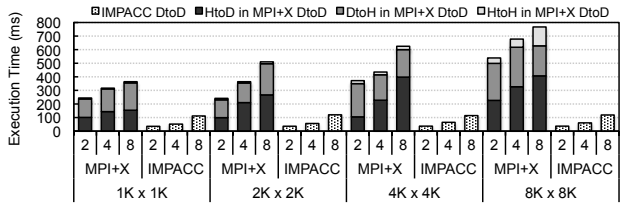

Figure 14: Device-to-Device Communication Time Breakdown for Jacobi in PSG. The x-axis represents the number of tasks (2, 4, and 8) and the number of elements in mesh.

Figure 14 shows the amount of total execution time for device-to-device communication between the tasks of Jacobi in PSG. As shown in Figure 6 (c), a device-to-device communication in IMPACC needs a single direct memory transfer between the devices via PCIe (noted as IMPACC DtoD in Figure 14). In MPI+OpenACC (noted as MPI+X), on the other hand, a device-to-device communication needs additional involvement of the host CPU and system memory (noted as HtoD, DtoH, and HtoH in MPI+X D2D). IMPACC significantly decreases device-to-device communication overhead, and thus allows for better performance.

Figure 13 (e) shows the speedup numbers in Beacon. The higher bandwidth in the intra-node point-to-point communication across the tasks in IMPACC is hidden with the small number of tasks because the kernel execution time dominates its total execution time. However, as the number of tasks increases, the kernel workload for each task decreases while the total amount of communication for each node does not change. Therefore, the communication overhead occupies the main part of its total execution time with a large number of tasks. As shown in Figure 13 (e) with 16, 32, and 64 tasks, IMPACC shows better performance than MPI+OpenACC due to its reduced communication overhead.

However, the communication overhead in Jacobi with a large number of tasks dominates its total execution ultimately, and it leads to poor scalability as shown in Figure 13 (e) with 128 tasks. Figure 13 (f) shows the strong scalability over 128 tasks in Titan. It shows worse performance as the

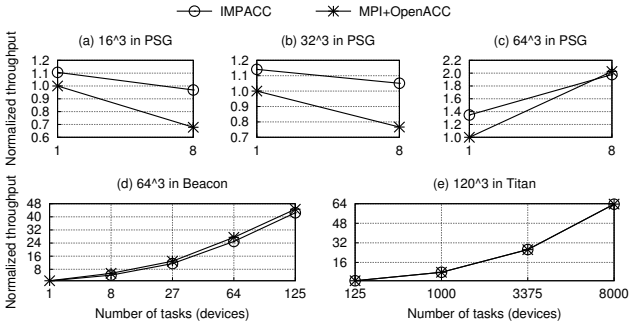

Figure 15: Performance Scaling of LULESH, normalized to MPI+OpenACC 1-task in PSG and Beacon, 125-tasks in Titan.

number of tasks increases due to its increasing communication overhead.

LULESH. LULESH is a shock hydrodynamics proxy application developed by Lawrence Livermore National Laboratory[33]. It solves the hydrodynamics equations on a staggered 3D spatial mesh. A task on the 3D sub-mesh computes on the volume of assigned elements ($O(N^3)$), and it transfers its surface elements in the mesh between its nearest neighbors in a Cartesian topology($O(N^2)$). And thus, computation-to-communication ratio is proportional to problem size ($O(N)$).

We run unmodified LULESH 2.0.2 MPI+OpenACC version[2] for both MPI+OpenACC and IMPACC, and thus all communications between tasks are host-to-host communications. Figure 15 shows the weak scalability for LULESH. Because LULESH requires that the number of tasks be a perfect cube, we vary the number of tasks to x^3 such as 1, 8, 27, 64, 125, 1000, 3375, and 8000. The graph titles in Figure 15 show the problem sizes for each task.

In a single node on PSG, IMPACC shows better performance than MPI+OpenACC due to the NUMA-friendly task-CPU mapping in a single task and message fusion technique that eliminates inter-process communication between tasks.

Beacon shows about 5% performance degradation in IMPACC compared to MPI+OpenACC. This comes from the IMPACC runtime overhead. There is no room for IMPACC to optimize the host-to-host internode communication. Also, the task threads shift their intra-node communication onto the communication thread by inserting message commands into the intra-node message queues. It introduces additional creating message commands overhead and queue scheduling overhead, resulting in slightly lower performance. For large problem sizes, since kernel execution time dominates the total execution time, both IMPACC and MPI+OpenACC in Titan show similar performance and achieve almost linear scalability.

5. RELATED WORK

The primary goal of HPC programming models is to provide the programmers with easy programming and high performance. There are some papers that propose novel programming models for accelerator systems to achieve both of them. StarSs and OmpSs[14, 17, 23] present a directive-based programming model for various accelerators such as Cell BE and multi-GPU. ADSM[27] provides a data-centric programming model that presents a shared address space between CPUs and accelerators. Haidar *et al.*[31] design algorithms and a programming model for dense linear algebra in accelerator systems. However, these approaches are limited to single node systems.

There have been efforts to provide novel unified programming models for heterogeneous architectures in both single-node and multi-node systems. Bueno *et al.*[19] present an implementation of OmpSs for GPU clusters. The runtime system automatically schedules the annotated codes with task directives to local or remote GPUs in the cluster. Phalanx[26] provides a synchronous task model supporting both coarse-grained and fine-grained parallelism. It also uses an IMPACC-like memory model to present a global address interface on multi-node systems. StarPU[12, 13] is a task programming library for heterogeneous architectures. Applications written in StarPU's language extensions submit computational tasks, with target device implementations, and StarPU schedules these tasks and associated data transfers on available devices. Cashmere[32] is a programming system for heterogeneous manycore clusters. It includes a framework to write and optimize kernels for different types of manycore devices, and it delivers automatic load balancing among them. HAM-Offload[40] defines a unified API to offload work to local and remote Intel Xeon Phi coprocessors in cluster environments. However, rewriting and restructuring the legacy HPC applications, that could contain tens or hundreds of thousands of lines, with new programming models requires a significant effort.

Some papers provide techniques that extend the target platform of existing programming models to heterogeneous accelerator clusters. SnuCL[34], LibWater[30], and rCUDA[22] extend the platform model of OpenCL and CUDA to the distributed clusters, respectively. However, their master-slave execution models do not scale well to the large-scale clusters due to the bottleneck of the centralized scheduling. Kwon *et al.*[36] presents a fully automated compiler-runtime system that translates and executes regular and repetitive OpenMP programs on clusters. However, their approach has some weakness, compared with hand-tuned MPI, due to the intrinsic limitations of static compiler techniques and runtime system overhead.

Exploiting shared memory parallelism in MPI implementations has been proposed by some work. HMPI[24] and ownership passing[25] share memory across MPI processes executing on the same node. TMPI[47] and MPI endpoint[21] relax the one-to-one relationship between MPI ranks and processes, and it enables the threads to act as MPI ranks. KNEM[28] provides MPI implementations with a scalable interface for performing kernel-assisted direct data transfers between local processes. However, to our best knowledge, no work has so far been published to exploit shared memory parallelism in MPI to expedite the accelerator data transfers.

MPI-ACC[10], MT-MPI[46], and some GPU-aware MPI implementations[5, 6, 42, 50] integrate MPI and accelerator programmings such as CUDA and OpenCL. However, they do not solve the synchronization issues between MPI and accelerator programmings, and/or inter-process communication overhead across MPI tasks as our approach does.

6. CONCLUSIONS

In this work, we propose IMPACC, a tightly integrated MPI+OpenACC framework for heterogeneous accelerator clusters. IMPACC starts with the observation that current MPI+OpenACC model has some inefficiencies and complexities to orchestrate two orthogonal programming models. IMPACC solves these problems by exploiting shared memory parallelism and the tight integration of MPI and OpenACC. IMPACC also proposes a new OpenACC directive extension to exploit the features of IMPACC, such as unified MPI communication routines, unified OpenACC activity queue, and node heap aliasing technique, easily and portably. We implement IMPACC and evaluate its performance with three accelerator systems. The evaluation results demonstrate that IMPACC achieves easier programming, higher performance, and better scalability than current MPI+OpenACC model.

7. ACKNOWLEDGMENTS

This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This material is based upon work supported by the National Science Foundation under Grant Number 1137097 and by the University of Tennessee through the Beacon Project. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation or the University of Tennessee. The authors would like to thank NVIDIA for providing access to their PSG Cluster.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy. The United States Government retains and the publisher, by accepting the article for publication, acknowledges that the United States Government retains a non-exclusive, paid-up, irrevocable, world-wide license to publish or reproduce the published form of this manuscript, or allow others to do so, for United States Government purposes. The Department of Energy will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan(http://energy.gov/downloads/doe-public-access-plan).

8. REFERENCES

[1] *Titan - Oak Ridge Leadership Computing Facility/Oak Ridge National Laboratory*, 2012. https://www.olcf.ornl.gov/titan/.

[2] *Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH) - Co-design at Lawrence Livermore National Laboratory*, 2013. https://codesign.llnl.gov/lulesh.php.

[3] *The OpenACC Application Programming Interface*, 2013. http://openacc.org.

[4] *AMD FirePro DirectGMA*, 2014. http://developer.amd.com/tools-and-sdks/graphics-development/firepro-sdk/firepro-directgma-sdk/.

[5] *Mellanox OFED GPUDirect RDMA*, 2014. http://www.mellanox.com/related-docs/prod_software/PB_GPUDirect_RDMA.PDF.

[6] *NVIDIA GPUDirect*, 2014. https://developer.nvidia.com/gpudirect.

[7] *MPICH: High-Performance Portable MPI*, 2015. http://www.mpich.org/.

[8] *PSG Cluster - NVIDIA Technology Center*, 2015. http://www.nvidia.com/nvtechcenter/.

[9] *TOP500 Supercomputer Sites*, 2015. http://www.top500.org/.

[10] A. M. Aji, J. Dinan, D. Buntinas, P. Balaji, W.-c. Feng, K. R. Bisset, and R. Thakur. MPI-ACC: An Integrated and Extensible Approach to Data Movement in Accelerator-based Systems. In *Proceedings of the 2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, HPCC '12, pages 647–654, 2012.

[11] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[12] C. Augonnet, O. Aumage, N. Furmento, R. Namyst, and S. Thibault. StarPU-MPI: Task Programming over Clusters of Machines Enhanced with Accelerators. In *Proceedings of the 19th European Conference on Recent Advances in the Message Passing Interface*, EuroMPI'12, pages 298–299, 2012.

[13] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 863–874, 2009.

[14] E. Ayguadé, R. M. Badia, F. D. Igual, J. Labarta, R. Mayo, and E. S. Quintana-Ortí. An Extension of the StarSs Programming Model for Platforms with Multiple GPUs. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 851–862, 2009.

[15] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS Parallel Benchmarks - Summary and Preliminary Results. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, SC '91, pages 158–165, 1991.

[16] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, and R. Thakur. Fine-Grained Multithreading Support for Hybrid Threaded MPI Programming. *Int. J. High Perform. Comput. Appl.*, 24(1):49–57, Feb. 2010.

[17] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta. CellSs: A Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06, 2006.

[18] R. G. Brook, A. Heinecke, A. B. Costa, P. Peltz, V. C. Betro, T. Baer, M. Bader, and P. Dubey. Beacon: Deployment and Application of Intel Xeon Phi Coprocessors for Scientific Computing. *Computing in Science Engineering*, 17(2):65–72, Mar 2015.

[19] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta. Productive Programming of GPU Clusters with OmpSs. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 557–568, 2012.

[20] F. Cappello and D. Etiemble. MPI Versus MPI+OpenMP on IBM SP for the NAS Benchmarks. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, 2000.

[21] J. Dinan, P. Balaji, D. Goodell, D. Miller, M. Snir, and R. Thakur. Enabling MPI Interoperability Through Flexible Communication Endpoints. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 13–18, 2013.

[22] J. Duato, A. J. Pena, F. Silla, J. C. Fernandez, R. Mayo, and E. S. Quintana-Orti. Enabling CUDA Acceleration Within Virtual Machines Using rCUDA. In *Proceedings of the 2011 18th International Conference on High Performance Computing*, HIPC '11, pages 1–10, 2011.

[23] V. Elangovan, R. Badia, and E. Parra. Ompss-opencl programming model for heterogeneous systems. In *Languages and Compilers for Parallel Computing*, volume 7760 of *Lecture Notes in Computer Science*, pages 96–111. 2013.

[24] A. Friedley, G. Bronevetsky, T. Hoefler, and A. Lumsdaine. Hybrid MPI: Efficient Message Passing for Multi-core Systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 18:1–18:11, 2013.

[25] A. Friedley, T. Hoefler, G. Bronevetsky, A. Lumsdaine, and C.-C. Ma. Ownership Passing: Efficient Distributed Memory Programming on Multi-core Systems. In *Proceedings of the*

18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '13, pages 177–186, 2013.

[26] M. Garland, M. Kudlur, and Y. Zheng. Designing a Unified Programming Model for Heterogeneous Machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 67:1–67:11, 2012.

[27] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-m. W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proceedings of the Fifteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 347–358, 2010.

[28] B. Goglin and S. Moreaud. KNEM: A Generic and Scalable Kernel-assisted Intra-node MPI Communication Framework. *J. Parallel Distrib. Comput.*, 73(2):176–188, Feb. 2013.

[29] M. Gorman. *Understanding the Linux Virtual Memory Manager*. Prentice Hall PTR, 2004.

[30] I. Grasso, S. Pellegrini, B. Cosenza, and T. Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 161–172, 2013.

[31] A. Haidar, C. Cao, A. Yarkhan, P. Luszczek, S. Tomov, K. Kabir, and J. Dongarra. Unified Development for Mixed Multi-GPU and Multi-coprocessor Environments Using a Lightweight Runtime Environment. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 491–500, 2014.

[32] P. Hijma, C. J. H. Jacobs, R. V. v. Nieuwpoort, and H. E. Bal. Cashmere: Heterogeneous Many-Core Computing. In *Proceedings of the 29th of IEEE International Parallel and Distributed Processing Symposium*, IPDPS'15, pages 135–145, May 2015.

[33] I. Karlin. LULESH Programming Model and Performance Ports Overview. Technical Report LLNL-TR-608824, Lawrence Livermore National Laboratory, Dec 2012.

[34] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12, pages 341–352, 2012.

[35] G. Krawezik. Performance Comparison of MPI and Three OpenMP Programming Styles on Shared Memory Multiprocessors. In *Proceedings of the Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '03, pages 118–127, 2003.

[36] O. Kwon, F. Jubair, R. Eigenmann, and S. Midkiff. A Hybrid Approach of OpenMP for Clusters. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 75–84, 2012.

[37] S. Lee and J. S. Vetter. Early Evaluation of Directive-based GPU Programming Models for Productive Exascale Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 23:1–23:11, 2012.

[38] S. Lee and J. S. Vetter. OpenARC: Open Accelerator Research Compiler for Directive-based, Efficient Heterogeneous Computing. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '14, pages 115–120, 2014.

[39] J. Meredith, P. Roth, K. Spafford, and J. Vetter. Performance Implications of Nonuniform Device Topologies in Scalable Heterogeneous Architectures. *IEEE Micro*, 31(5):66–75, Sep 2011.

[40] M. Noack, F. Wende, T. Steinke, and F. Cordes. A Unified Programming Model for Intra- and Inter-node Offloading on Xeon Phi Clusters. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 203–214, 2014.

[41] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe. Portable Performance on Heterogeneous Architectures. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 431–444, 2013.

[42] S. Potluri, D. Bureddy, K. Hamidouche, A. Venkatesh, K. Kandalla, H. Subramoni, and D. K. D. Panda. MVAPICH-PRISM: A Proxy-based Communication Framework Using InfiniBand and SCIF for Intel MIC Clusters. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 54:1–54:11, 2013.

[43] R. Rabenseifner, G. Hager, and G. Jost. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, 2009.

[44] S. L. Scott. Synchronization and Communication in the T3E Multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, pages 26–36, 1996.

[45] S. Seo, G. Jo, and J. Lee. Performance Characterization of the NAS Parallel Benchmarks in OpenCL. In *Proceedings of the 2011 IEEE International Symposium on Workload Characterization*, IISWC '11, pages 137–148, 2011.

[46] M. Si, A. J. Peña, P. Balaji, M. Takagi, and Y. Ishikawa. MT-MPI: Multithreaded MPI for Many-core Environments. In *Proceedings of the 28th ACM International Conference on Supercomputing*, ICS '14, pages 125–134, 2014.

[47] H. Tang, K. Shen, and T. Yang. Compile/Run-time Support for Threaded MPI Execution on Multiprogrammed Shared Memory Machines. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '99, pages 107–118, 1999.

[48] D. Tarditi, S. Puri, and J. Oglesby. Accelerator: Using Data Parallelism to Program GPUs for General-purpose Uses. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 325–335, 2006.

[49] J. S. Vetter. *Contemporary High Performance Computing: From Petascale Toward Exascale*. Chapman & Hall/CRC, 2013.

[50] H. Wang, S. Potluri, M. Luo, A. K. Singh, S. Sur, and D. K. Panda. MVAPICH2-GPU: Optimized GPU to GPU Communication for InfiniBand Clusters. *Comput. Sci.*, 26(3-4):257–266, June 2011.

[51] X. Wu and V. Taylor. Performance Characteristics of Hybrid MPI/OpenMP Implementations of NAS Parallel Benchmarks SP and BT on Large-scale Multicore Supercomputers. *SIGMETRICS Perform. Eval. Rev.*, 38(4):56–62, 2011.