Extending MPI to Accelerators

Jeff A. Stuart
University of California, Davis
stuart@cs.ucdavis.edu

Pavan Balaji
Argonne National Laboratories
balaji@mcs.anl.gov

John D. Owens
University of California, Davis
jowens@ece.ucdavis.edu

ABSTRACT

Current trends in computing and system architecture point towards a need for accelerators such as GPUs to have inherent communication capabilities. We review previous and current software libraries that provide pseudo-communication abilities through direct message passing. We show how these libraries are beneficial to the HPC community, but are not forward-thinking enough. We give motivation as to why MPI should be extended to support these accelerators, and provide a road map of achievable milestones to complete such an extension, some of which require advances in hardware and device drivers.

1. INTRODUCTION

The world of High-Performance Computing (HPC) is no longer CPU-only, as CPUs are largely hitting power and performance walls. To first order, CPU clocks will not increase in frequency for the foreseeable future. This has caused the HPC community explore the use of accelerators such as the graphics-processing unit (GPU).

Accelerators have an inherent problem in computing: they are slave devices. The CPU controls these devices explicitly, and the accelerators do nothing but computation, own-memory access, and participate in DMA transfers. Even though accelerators often reside on the PCI-e bus, many have no peer-to-peer capabilities, contradicting a primary design goal of PCI-e. Another byproduct of being slave devices is a lack of inherent message-passing capabilities. In the modern compute node, only CPUs have the inherent ability to communicate with other compute devices via MPI, whether they be on the same node or on different nodes. We see this as a deficiency and complication in programming accelerators. Many algorithms port naturally to different types of accelerators, except that they often contain small fragments that require communication with other compute devices. Modern practices dictate that at these points, users exit their computational kernel to allow the CPU to issue communication requests.

If accelerators were to have message-passing capabilities, we feel the obvious API is the Message-Passing Interface (MPI) [2]. MPI is an industry-wide standard and is supported by virtually every high-speed interconnect manufacturer. Most HPC developers prefer to use MPI to develop applications with low-level communication requirements (i.e. at a lower level than global arrays or MapReduce). As accelerators do not inherently have communication capabilities, they do not and cannot fully support MPI.

Our aim is to address MPI in the accelerator era, and more specifically the issues of supporting MPI on accelerators, and we use the GPU as an example. We wish to provide a roadmap to port MPI to accelerators and do so by expanding upon previous work in the area. As these ports are not straightforward, we have to be careful in how we choose to extend MPI, what support we offer initially on each accelerator, and try to focus on the common aspects of accelerators. The end goal of this paper is to explore how to include accelerators as first-class MPI devices. We also hope to spur vendor support — to the point where accelerators natively support efficient and high-performance MPI implementations.

2. PREVIOUS WORK

MPI is the most widely used communication library for HPC. It combines simple-to-use communication routines with advanced features like one-sided messaging. It has wide-spread support from virtually all high-speed interconnect manufacturers, making it a natural choice for HPC applications due to its high level of portability.

MPI is a low-level a library on which higher-level models like global arrays or MapReduce can be built. Because of the abstraction of higher-level libraries like these, they lack the fine-grained control that MPI offers. We feel that bringing MPI to accelerators is the first in a series of necessary steps in making accelerators first-class citizens in HPC.

In order to leverage MPI on accelerators, researchers have implemented several different extensions to, and variations of, MPI. cudaMPI [3] provided an MPI-like interface that handled the inherent memory copies from the GPU to the CPU before the CPU executed an MPI call. However, thanks to 0-copy memory[1] introduced in CUDA 3 [4], cudaMPI is redundant. GAMPI [1] took a different approach by making GPUs logically separate MPI targets and providing default communicators besides *MPI_COMM_WORLD*, such as an all-CPU communicator, an all-GPU communicator, and a

[1] 0-copy memory is pinned system memory that is accessible by both the CPU and accelerators.

ASBD 11, October 10, 2011, Galveston Island
Copyright 2011 ACM 978-1-4503-1439-8/12/04 ...$10.00.

local-GPUs communicator. However, as with cudaMPI, the CPU was explicitly responsible for initiating communication.

Stuart and Owens implemented DCGN [6], a more in-depth GPU message-passing library. DCGN provides the point-to-point and collective capabilities of MPI directly to CUDA kernels. While the CPU still must execute MPI calls, DCGN hides this from the user and provides the illusion of the GPU sourcing and sinking communication. DCGN manages all the necessary PCI-e memory transfers behind the scenes[2]. DCGN also introduces the notion of slots in order to deal with ranks on a GPU. Because of the unique execution strategy of the GPU, the user specifies how many ranks per GPU they would like, as opposed to forcing an arbitrary number (e.g. one per GPU, or one per thread processor).

The biggest problem we see with cudaMPI, GAMPI, and DCGN is simply the lack of hardware and driver support. Only recently has a GPU vendor announced plans to increase the GPU-NIC throughput (as is the case with the new GPU-Direct technology between Infiniband-manufacturer Mellanox and GPU-manufacturer NVIDIA). GPU-Direct technology removes the need for an extra buffer copy by the CPU when transmitting data to or from a GPU, but the CPU is still required to initiate the transfer. GPU-Direct and other advances make much of cudaMPI and GAMPI irrelevant. And as the CPU must always execute MPI calls itself, DCGN still would incur overhead in CPU cycles and also possibly data-transit time.

Some of the troubles with DCGN are resident in another common programming model: MPI with threads. MPI is designed to be used with processes, but as the HPC world migrates to multi-core and multi-processor clusters, developers are now leveraging CPU threads. For example, in a multi-core, multi-socket node, a user might start one MPI process per socket and then one thread per additional core in the socket. All threads in a process share a rank, meaning in an MPI-without-threads implementation, communication must be explicitly handled by one thread. In the case of an MPI-with-threads implementation, tags and/or other forms of disambiguation must be used. We draw upon the latter model for inspiration later in this paper.

3. ROADMAP

In an ideal situation, we would like to extend MPI to all accelerators. Thus we propose changes in this section that, while focused on the GPU because it is the most common accelerator in use in HPC right now, are applicable to all accelerators. We feel that the challenges in porting MPI to the GPU will aid in porting to other accelerators. We focus primarily on CUDA but note that our changes are equally applicable to OpenCL. This section outlines the technical and philosophical challenges we must overcome to extend MPI to accelerators and the GPU specifically.

3.1 MPI Ranks and Accelerators

MPI provides ranks in MPI_COMM_WORLD only to processes. DCGN provides a variable number of ranks (slots) to both processes and accelerators. GAMPI provides a rank to each accelerator, though it does not allow the accelerator to source or sink communication. cudaMPI allows accelerator memory to be used as the source and sink for communication, but does not give the accelerator a rank and does not allow the accelerator to execute MPI calls (the CPU must do so). The space of possibilities for how to assign ranks is very large, and we are still determining which solution is best. Following is a list of the tenable choices for how to assign ranks to accelerators, along with their effects on the interface, implementations, and usage model of MPI.

3.1.1 No Ranks for Accelerators

This is the current model supported by MPI. Under this model, no changes to the MPI implementation must be made, but users that wish to use accelerators cannot make MPI calls from within accelerator kernels. With a unified address space (as is offered by the most recent CUDA toolkit), pointers to accelerator memory can be passed to MPI functions.

Changes to MPI.

With no ranks given to accelerators, there are little to no changes to the MPI. In some cases, the MPI implementation might have extensions to handle data transfers between system memory and the accelerator. However, we do not feel this is a worthwhile extension.

Application Usage Model.

As this is the default model for MPI, the application usage model remains unchanged.

MPI Implementation Implications.

As there are no changes to the MPI, the implementation implications are nil.

3.1.2 Several Ranks per Accelerator Context

This is the model used by DCGN. In it, each accelerator context is given a variable number of ranks (slots), specified by the user at runtime. Accelerators are controlled by library-spawned threads; GPU kernels can execute communication requests but the CPU threads controlling them cannot.

Several ranks per accelerator context make sense because many accelerators have more than one core. There does not exist a “one-size-fits-all” best mapping of ranks to threads/ blocks/SMs on a GPU, nor on a cell. However, MPI allows tags (DCGN does not), hence more than one rank might simply create extra overhead for an MPI implementation.

Changes to MPI.

The changes to MPI in this case are significant. First and foremost, MPI transitions from a process-based model to a thread-based model. Second, the MPI must do one of two things. Either have a method for an accelerator thread to dynamically bind to one rank from a subset of ranks, or have every MPI call require an extra argument, the source rank. We require this because every thread in the accelerator can make MPI calls, but MPI has no way of knowing from which of the many possible source ranks the call originates.

Application Usage Model.

On the CPU, the usage model deviates slightly. First and foremost, an MPI process now has many threads: one thread

[2] This has been made redundant thanks to 0-copy, as well as other advances in CUDA 4 [4] such as a unified address space between all CPUs and GPUs on a node.

per accelerator, plus the main thread. The accelerator-controlling threads are spawned from within MPI_Init. They exist solely to execute accelerator kernels, though they can use the rank given to the CPU process, just like in a multi-threaded MPI implementation. From within the main CPU thread and the accelerator kernels, the usage model stays the same: simply execute code and when an MPI call is made, the library will handle it accordingly.

Kernels are executed in the usual way provided by their own API (i.e. the MPI does not provide a wrapper for launching kernels), and are always asynchronous. However, in order for MPI to monitor for communication calls from the accelerator, the user must call an MPI-provide polling or blocking function while kernels are running.

MPI Implementation Implications.

The implications to MPI are significant. First and foremost, either new functions for binding accelerator threads to ranks must be created, or every MPI call must have its signature modified to allow for a source rank to be specified. Second, the mandatory use of user-space threads deviates significantly from the MPI model. An MPI implementation must be thread-safe to fully support this execution model. Third, new functions must be introduced to monitor for GPU communications explicitly, thus not guaranteeing progress within a kernel unless user code for the CPU is written properly.

3.1.3 One Rank per Accelerator Context

This is the analog of one rank per MPI process. Each accelerator context is given a rank. Significant modifications to the underlying implementation must be made, and we further modify the MPI model.

Changes to MPI.

At the highest level, the changes to MPI involve a new type of process: MPI will now also create processes to control accelerators. These processes may or may not have any communication capabilities on the CPU (discussed further below), meaning MPI could issue a runtime error if a function like *MPI_Isend* is called from one of these CPU processes. Instead, MPI gives a rank to the accelerator context and the kernels on the accelerator execute MPI function.

Application Usage Model.

On job startup, **every** process calls MPI_Init[3], and MPI_Init still has collective semantics. Which (if any) accelerators the current process controls is also passed implicitly to MPI_Init. To allow users to query which (again, if any) accelerators they own, we provide a new function *MPI_Controls_accelerators*. The user can branch based on the outcome (much like master-slave MPI jobs based on a processes rank), using any accelerators it may own.

To launch kernels, a process simply invokes the kernel using the API provided by the accelerator vendor (again, the MPI does not provide wrappers for launching kernels). To monitor for communication requests from the accelerator, we suggest that the MPI implementation simply create a user-invisible thread and use a polling mechanism similar to the GPU-to-CPU callbacks mechanism [5] implemented by Stuart et al.

MPI Implementation Implications.

With this model, MPI implementations must support two implementations of all MPI functions (with the exception of MPI_Init and MPI_Finalize): one for CPU processes, and one for accelerator kernels. MPI must also add the accelerator query function described above. As arguments, MPI_Controls_accelerators takes an integer pointer *deviceCount*, and two integer arrays *deviceTypes* and *deviceIDs*. *deviceCount* must be initially set to the capacity of the *deviceTypes* and *deviceIDs* arrays. If the value returned in *deviceCount* is zero, then this process owns no accelerators. If the value is less than zero, the arrays are not large enough to hold the return values, and must be resized to at least $-1 \times$ *deviceCount* and the user should then call this function again. If the value is greater than zero, this is how many accelerator contexts the current process owns. The types of each accelerator (e.g. NVIDIA GPU, IBM Cell, etc.) are stored in *deviceTypes*, and are identified via MPI constants (the names of which are yet to be determined). The device IDs are returned in *deviceIDs*, and correspond to the logical IDs of the devices, as enumerated by the appropriate device driver (e.g. with CUDA, the first GPU is device 0).

One technical problem is how to monitor for communication requests from kernels. We again propose the polling mechanism from DCGN and the GPU-to-CPU callbacks research.

Another technical problem is how the MPI should group accelerators into processes. For example, if a node has four accelerators, does the user want all four accelerators controlled by one process? Perhaps two processes each control one accelerator? Or two processes with one accelerator controlled by the first and the other three accelerators controlled by the second? By default, we think MPI should assign one accelerator per process. If the user wishes for a different configuration, the MPI implementation should provide a means to do so via the hosts file or perhaps a new "accelerators" file.

Regarding implementation implications, we have one more very important decision to make: whether we allow the CPU to use a rank from one of the accelerator contexts. If we decide against, we change the behavior of MPI in that MPI will create user-visible processes wherein the CPU cannot execute MPI calls, only the accelerator may. This requires the MPI to branch within every MPI function, or to a function pointer or jump table, to flag offending calls and raise an error. It also complicates issues for the user because if the CPU cannot communicaite, copying data to/from the accelerator may take extra time. There are only a limited number of CPU shared-memory pages and these pages do not necessarily work smoothly with accelerator and Infiniband drivers, thus making it non-trivial to share buffers, on a single node with an accelerator, between processes.

On the other hand, if we choose to allow CPU processes to share ranks with accelerator contexts, we shift the hardships from the user to the MPI implementation. We have a model wherein we encounter similar challenges to a multithreaded MPI implementation. We can have concurrent MPI calls from different threads of execution—in this case

[3] We initially thought it prudent to create a new type of init function (e.g. *MPI_Init_accelerators*), but decided against it because that would imply the process must know, before initializing MPI, if it will be an accelerator-controlling process, or a regular MPI process. Such information should not be required for a process before MPI initialization.

one or more threads will come from the CPU, and one or more will come from the accelerator. This creates significant complications for the MPI implementation because it must maintain matching order for all messages on the same communicator, tag, and source. Because MPI calls can come from multiple sources for the same rank, the implementation must perform a lot of coordination. Fortunately, there is already significant research for MPI with threads, and we can start from there when implementing this for a CPU and accelerator. The deadlock/application-hang problems associated with a multithreaded MPI implementation (e.g. two threads with the same rank executing an *MPI_Barrier*) are found here, but they are not unique to MPI with accelerators and thus the previous caveats and avoidance strategies are valid.

To further complicate matters when sharing a rank between a CPU process and an accelerator, if the CPU process owns multiple accelerator contexts, the implementation must choose which accelerator shares its rank with the CPU. We need more time to investigate an optimal strategy, and potential means for user input into this decision. A good default is to share with the first accelerator, that is the accelerator signified in *deviceIDs*[0].

3.1.4 New MPI Function(s) to Spawn Kernels

In this model, we use a new set of MPI functions to spawn MPI-capable kernels. New communicators are created and ranks assigned upon spawning, and all resources are reclaimed upon finalization of the kernels.

Changes to MPI.

This model offers a good balance between new capabilities and minimal changes to MPI. Accelerators, contexts, and kernels have no ranks initially. When a user wishes to launch a kernel on one or more accelerators, and to give those kernels communication abilities, we propose a new function similar to *MPI_Comm_spawn*, which we will tentatively call *MPI_Comm_spawn_kernels*. When this function is called, MPI launches the kernel on a set of accelerators and gives each kernel a rank within a new MPI_COMM_WORLD. Similar to MPI_Comm_spawn, the user may pass hints to the MPI as to where the kernels should be spawned.

If requested, an inter-communicator is provided by the MPI to allow communication between the CPU processes and the kernels. Upon completion of the final kernel, MPI destroys the spawned communicator and reclaims all resources. With this method, existing code in the MPI implementation can largely be left alone. Instead, only new functions are necessary. Also, users of accelerators need not worry about running a kernel at the right time to initialize MPI, finalize MPI, or handle collectives.

Application Usage Model.

The changes to the existing usage model is incremental. Applications which already use kernels may safely do so. If a user wishes to grant kernels MPI capabilities, they may do so via using the spawn function, passing a context for each accelerator into the spawn function. Otherwise kernels may still be launched in the usual manner. Only kernels executed via spawn functions will have MPI capabilities.

MPI Implementation Implications.

The implications to MPI implementations are less significant in this model. Again, implementations must support a polling mechanism to monitor for communication requests, and also support both CPU and accelerator versions of MPI functions. Ranks in communicators are associated with kernels, not with contexts, so it is fairly easy for MPI to monitor for the death of a kernel and reclaim resources.

3.2 MPI Execution Environment and Node and Process Management

We intend to make many changes to the MPI runtime environment and the way MPI treats processes, as well as make changes to the the MPI job launcher (e.g. mpirun). We need to change the environment to better handle the nature of accelerator communication requests and the manner in which accelerator drivers work. And we have to change the job launcher to account for changes in resources; we now have accelerators as well as CPUs.

MPI Processes and Accelerators.

Each accelerator has a unique driver, runtime, and execution environment. To talk about the interaction between MPI, runtimes, and execution environments, we must address each type of accelerator individually and then abstract away common properties to ease the burden of implementation. In this section, we specifically address the challenges presented with NVIDIA GPUs and CUDA but focus on the higher-level points that apply to all accelerators.

CUDA initializes and controls GPUs through a data structure called a *context*. In previous implementations of CUDA, a context was thread-specific, meaning only the thread that created the context could access the GPU. With CUDA 4.0, this is no longer the case: any thread can access any GPU safely. This new freedom makes it tempting to create a CPU MPI process, and have it spawn as many threads as there are GPUs to control each GPU. However, too many issues arise with the use of threads. The first is address spaces: any memory allocations by one thread implicitly affect the address space of all other threads in the same process. The second complication is with global variables. While frowned upon in development, many MPI applications utilize global variables but do not account for extra threads in their process space. Also, using threads in an MPI library forces the library to have layers of thread safety. Even more subtle complications exist, and these issues make us lean towards maintaining the use of processes for MPI computing primitives.

As MPI creates all processes for a job (including accelerator-controlling processes), and because MPI cannot implicitly monitor the accelerator for communication requests (it must use a polling mechanism similar to that of DCGN), we must create a simple model for users to accomplish both CPU and accelerator communication. We have come up with several options and discuss them later.

Provided Communicators.

At startup, MPI provides a default global communicator. Taking inspiration from GAMPI, we believe that other communicators are useful, specifically, an accelerator-only communicator, a CPU-only communicator, and two local-node only equivalents. The names of these new communicators are not of primary importance and thus have not yet been resolved. While the MPI forum had the chance to implement a local-CPU-only communicator and chose not to, we

	DCGN [6]	cudaMPI [3]	GAMPI [1]	MPIWA
CPU-GPU Buffer Sharing	Yes	Yes	Yes	Yes
GPU Ranks	Yes	No	Yes	Yes
Extra Communicators	No	No	Yes	Yes
CPU + GPU Collectives	Yes	Yes	Limited*	Yes
GPU Source/Sink Comm	Yes	No	No	Yes
Standard MPI Calls on the GPU	No	No	No	Yes
MPI2 Feature Support	No	Limited†	Limited†	Limited†

Table 1: Feature Support provided by the three major implementations of message passing on GPUs, and our proposed implementations.

* Only between CPUs and GPUs on the same node.

† Only the CPU can execute most MPI2 calls, and both the source and destination must be CPUs. We require further time to investigate the impact of the CUDA 4.0 unified address space on features such as one-sided communication.

believe that all these communicators, in the era of accelerators and multi-core, are quite sensible.

The specific names of these communicators are secondary and up for debate. Each of these communicators offer something of inherent value to MPI programmers, especially when dealing with collectives. Of course, the accelerators do not make sense under certain implementations, which we discuss below.

Collectives.

Many MPI implementations use optimized collective routines that take advantage of shared-memory APIs for ranks on a single node. Such optimizations are also useful for accelerators. Any optimizations that exist for CPU ranks on a single machine can be extended/modified to also include, or be explicitly for, accelerators in the same node.

Sourcing and Sinking of Communication.

For implementations that support sourcing and sinking of communication by an accelerator, we believe it best to support such via accelerator-to-CPU callbacks [5] (meaning a separate thread/process polls flags in device-accessible memory to look for communication requests). A method such as this is necessary until a time when accelerator and NIC vendors give native MPI capabilities to the hardware. Thanks to CUDA 4, application developers can use native GPU memory or 0-copy memory in their kernels so the MPI implementation can poll each with little effort. The tradeoffs in overhead in both can impact application performance, thus the one to use should be chosen carefully.

3.3 Non-Persistent Accelerator Kernels

Many accelerator models are task-based in a manner similar to OpenMP, wherein a computational region spawns workers (e.g. threads on the GPU) that do some work and disappear. They are not persistent and with some implementations, this gives rise to logical deadlocks and/or application hangs, specifically when an accelerator is inactive but needed for a collective or synchronous communication. We feel that such situations are not an issue with the underlying MPI implementation though. It is quite easy to write a CPU-only MPI application with the same problem (e.g. a missing barrier call in a certain application which causes all other nodes to block indefinitely), and thus we feel it is the responsibility of the application developer to ensure that the appropriate kernels run as necessary to perform communication.

3.4 Comparison with Existing Implementations

To give an overview of the feature set given by previous implementations and by our proposed implementations, we present Table 1. The rows represent important features we feel are necessary for a successful MPI implementation on accelerators. None of the existing implementations provides the full range of features. The fourth column represents the characteristics of our proposal, which we call MPI with Accelerators (MPIWA).

4. REFERENCES

[1] A. Athalye, N. Baliga, P. Bhandarkar, and V. Venkataraman. GPU aware MPI (GAMPI) - a CUDA-based approach. Technical report, University of Texas, Austin, 2010.

[2] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. MIT Press, 1999.

[3] O. S. Lawlor. Message passing for GPGPU clusters: CudaMPI. In *IEEE International Conference on Cluster Computing and Workshops*, Oct. 2009.

[4] NVIDIA Corporation. NVIDIA CUDA programming guide. `http://developer.download.nvidia.com/compute/cuda/4_0_rc2/toolkit/docs/CUDA_C_Programming_Guide.pdf`, July 2011.

[5] J. A. Stuart, M. Cox, and J. D. Owens. GPU-to-CPU callbacks. In *Euro-Par 2010 Workshops: Proceedings of the Third Workshop on UnConventional High Performance Computing (UCHPC 2010)*, volume 6586 of *Lecture Notes in Computer Science*, pages 365–372. Springer, July 2011.

[6] J. A. Stuart and J. D. Owens. Message passing on data-parallel architectures. In *Proceedings of the 23rd IEEE International Parallel and Distributed Processing Symposium*, May 2009.