Achieving Computation-Communication Overlap with Overdecomposition on GPU Systems

Jaemin Choi*, David F. Richards†, Laxmikant V. Kale*
*Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois
†Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California
Email: {jchoi157,kale}@illinois.edu, richards12@llnl.gov

***Abstract*—The landscape of high performance computing is shifting towards a collection of multi-GPU nodes, widening the gap between on-node compute and off-node communication capabilities. Consequently, the ability to tolerate communication latencies and maximize utilization of the compute hardware are becoming increasingly important in achieving high performance. Overdecomposition has been successfully adopted on traditional CPU-based systems to achieve computation-communication overlap, significantly reducing the impact of communication on application performance. However, it has been unclear whether overdecomposition can provide the same benefits on modern GPU systems. In this work, we address the challenges in achieving computation-communication overlap with overdecomposition on GPU systems using the Charm++ parallel programming system. By prioritizing communication with CUDA streams in the application and supporting asynchronous progress of GPU operations in the Charm++ runtime system, we obtain improvements in overall performance of up to 50% and 47% with proxy applications Jacobi3D and MiniMD, respectively.**

***Index Terms*—computation-communication overlap, overdecomposition, asynchronous task-based runtime, GPU computing**

I. Introduction

Employment of GPUs as accelerators in today's high performance computing systems has played a significant role in the rapid increase in computational power, ushering in the exascale era. A single compute node of the Summit [1] supercomputer at Oak Ridge Leadership Computing Facility (OLCF) boasts a theoretical peak performance of over 40 TFLOPS with its six NVIDIA Tesla V100 GPUs. What is often disregarded, however, is that the single-node computational power is greatly outpacing the inter-node network performance. Comparing Summit to OLCF's former leadership system, Titan [2], the compute performance has multipled more than 28 times (1.4 TFLOPS vs. 40 TFLOPS) while the interconnect bandwidth has increased less than fourfold (6.4 GB/s vs. 23 GB/s).

The growing gap between on-node compute and off-node communication capabilities can be tackled from at least two different directions: 1) improving communication performance with optimizations in the software stack and better utilization of the hardware support (e.g. SHARP [3] for collectives, hardware tag-matching), and 2) reducing the impact of communication on overall performance by overlapping computation and communication. In this work, we focus on the latter in the context of GPU systems and seek to improve the performance of GPU-accelerated applications by achieving computation-communication overlap with overdecomposition.

Overdecomposition is an approach of decomposing the problem domain into logical sub-domains (units of work and/or data), often creating many more such sub-domains than the number of available processors. In addition to benefits such as better cache utilization, dynamic load balancing, and fault tolerance [4], overdecomposition facilitates computation-communication overlap by overlapping computation of a sub-domain with communication of another. While these merits have been well observed on traditional CPU-based systems, overdecomposition and its impact on performance have not been thoroughly explored on modern GPU systems. We apply overdecomposition to GPU-accelerated proxy applications using the Charm++ parallel programming system [4], and address critical issues in the application as well as the underlying runtime system to achieve computation-communication overlap. We show that a moderate degree of overdecomposition provides a substantial improvement in performance despite the potential drawbacks of finer-grained GPU kernels such as kernel launch and execution overheads.

The primary contributions of this work are as follows:

- Prioritization of communication with multiple CUDA streams in GPU-accelerated proxy applications to maximize computation-communication overlap
- Mechanisms to enable asynchronous progress of GPU operations in scheduler-driven task runtimes
- Weak and strong scaling performance analysis of overdecomposed proxy applications on leadership-class systems

II. Background

A. Overdecomposition

As briefly explained in Section I, overdecomposition allows the application programmer to divide the problem domain into work and data units unconstrained by the number of available processors. Figure 1 compares a typical decomposition scheme in MPI and overdecomposition with a factor of four. Overdecomposition decouples computational work and data of the application from hardware resources, empowering the runtime system to control the mapping of these work units to the available processors. This can also benefit programmer productivity as the units of decomposition become first-class citizens of the program and can be addressed with logical names.

978-0-7381-1074-5/20/$31.00 ©2020 IEEE
DOI 10.1109/ESPM251964.2020.00006

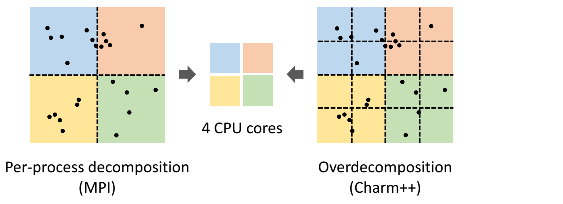


Fig. 1. Comparison of a typical decomposition in MPI and overdecomposition in Charm++. Particles are scattered across the two-dimensional global grid. MPI decomposition assigns one sub-domain to each process (a CPU core in pure MPI), whereas overdecomposition assigns four smaller sub-domains to each PE (generally a single core).

In regard to performance, an important benefit obtainable with overdecomposition is overlap of computation and communication. With multiple work units assigned to each processor, a work unit can perform computation while another mapped to the same processor is waiting for communication. This requires a high degree of asynchrony, however, which can be achieved with asynchronous message-driven execution as described in Section II-B2. To achieve computation-communication overlap on GPU systems, it is crucial to additionally support asynchronous progress of GPU operations with minimal delay. This is discussed in detail in Section III-B.

B. Charm++ Parallel Programming System

1) Overdecomposition: The Charm++ parallel programming system [4] realizes overdecomposition, in addition to other design principles such as migratability and asynchrony, as a C++-based approach to writing parallel programs. The problem domain is decomposed using special C++ objects called *chares* and an indexed collection of chares called *chare arrays*. Typically there are many more chares than the number of processors, empowering the underlying runtime system with more control over their execution. For example, the spatial domain in Figure 1 is overdecomposed into a 2D chare array of $4 \times 4 = 16$ chares that are mapped to four CPU cores. Following the object-oriented programming paradigm of C++, each chare encapsulates data and methods that perform operations on data. For instance, pairwise force calculation of particles in Figure 1 can be implemented as a *chare method* using particles' positions and mass as data.

2) Asynchronous message-driven execution: The set of processors selected by the user for parallel execution in Charm++ are referred to as *processing elements* (PEs). A PE generally corresponds to a CPU core on which a scheduler is executed, but it could also be a multi-threaded process running on multiple cores, where a single scheduler manages the multi-threaded execution. The scheduler is governed by an asynchronous message-driven execution model, where a chare is scheduled for execution on a PE when a message for it is received. This execution of a chare's method corresponds to a *task* in Charm++. A message queue associated with each scheduler stores the incoming messages, and the scheduler picks up a message from the queue when it is free, ordinarily in FIFO order. Communication in Charm++ is asynchronous as the sender chare does not block for any reply or acknowledgement from the receiver, and incoming messages are asynchronously stored in the message queue. Figure 2 illustrates an execution pattern with asynchronous message-driven execution in Charm++.

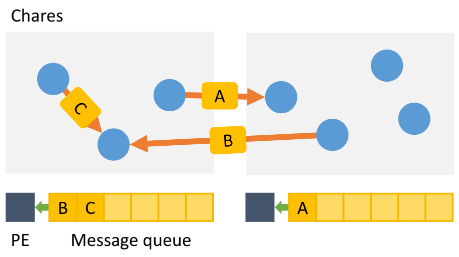


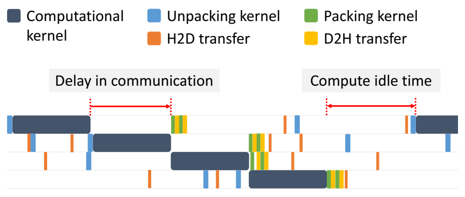
Fig. 2. Asynchronous message-driven execution in Charm++. Chares exchange messages which are stored in the message queue and picked up by the scheduler on each PE.

3) Computation-communication overlap: When computation phases are separated by communication as in bulk-synchronous models such as MPI, time spent in communication directly translates into idle time that affects the overall execution. This issue can be mitigated by performing an independent computation during the communication phase, but resources can still remain idle if blocking communication calls are invoked before messages arrive [5].

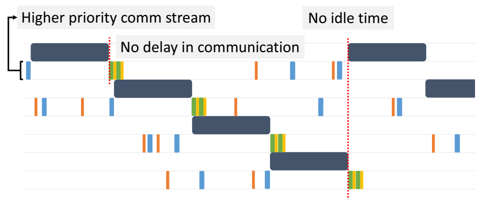
Overdecomposition with asynchronous message-driven execution provides a natural remedy to this problem. While a chare is waiting for communication, another chare can be scheduled to perform computation, effectively hiding communication latency. Furthermore, the injection of messages into the network is spread throughout the execution, in contrast to traditional MPI applications where communication is often clustered at certain points of the execution timeline.

4) GPU execution: A Charm++ programmer may implement chare methods that offload work to the GPU using any GPU programming model suitable for the hardware, including CUDA for NVIDIA GPUs and HIP for AMD GPUs. When such GPU operations are executed synchronously, e.g. using `cudaStreamSynchronize`, there is no additional support needed from the runtime. However, this prevents computation-communication overlap as the Charm++ scheduler is blocked from handling incoming messages and executing methods of other chares on the same PE. We provide an API in the Charm++ runtime to address this issue, currently using CUDA for NVIDIA GPUs. Its design and implementation details are provided in Section III-B.

The following sections of the paper assume the use of NVIDIA GPUs and discuss GPU usage in terms of CUDA. Support for AMD and Intel GPUs in Charm++ are being developed to prepare for upcoming leadership-class systems. It should also be noted that for chares to communicate GPU data, explicit data transfers between the host and device are currently required as production-level support for direct GPU-GPU transfers in Charm++ is in progress.



(a) Single CUDA stream per chare. Communication is delayed by a computational kernel enqueued from another chare, causing idle time between iterations.



(b) Separate compute/communication CUDA streams per chare, with the communication stream given higher priority. Iterations continue without idle times in between.

Fig. 3. Execution timelines of Jacobi2D with four chares mapped to a single GPU.

III. Achieving Computation-Communication Overlap

In this section, we discuss important considerations in the application and the underlying runtime system when applying overdecomposition to achieve computation-communication overlap on GPU systems.

A. *Prioritizing Communication in the Application*

A straightforward method of integrating GPUs into an application is associating a CUDA stream with each work unit. GPU operations including kernel launches and data transfers enqueued in the same stream are guaranteed to execute in order. For MPI applications where a single process is used to manage each GPU, it is often sufficient to use the default stream. In a Charm++ application, each chare can maintain a separate stream to enforce dependencies between GPU operations performed by the same chare. Note that multiple chares can be mapped to use the same GPU device.

While assigning a single CUDA stream to each chare ensures that dependencies are observed between operations performed with the chare's data, it often impedes computation-communication overlap. The primary cause is the delay in communication-related operations, including host-device data transfers and associated packing/unpacking kernels, due to computational kernels offloaded from other chares.

Figure 3 depicts execution timelines of two different implementations of Jacobi2D, a simple Charm++ program that performs the Jacobi iteration in a two-dimensional grid. The global grid is overdecomposed into four chares on a single PE, which is mapped to a GPU. Each chare is responsible for a quadrant of the grid and performs halo exchanges with its two neighboring chares after the Jacobi update, which is repeated for a given number of iterations. The communication-related GPU operations for each neighbor comprise of a packing kernel and a D2H transfer enqueued after the main Jacobi update kernel, as well as a H2D transfer and a unpacking kernel that are enqueued once a halo message is received. Figure 3a shows the execution of the implementation that uses a single CUDA stream per chare to enqueue all computation, i.e. Jacobi update, and communication-related operations. Although there is some overlap of computation and communication, idle time is observed due to the delay in the execution of the communication-related operations (packing kernels and D2H transfers).

This issue can often be resolved by utilizing separate streams for compute and communication in each chare, as in Figure 3b, with the communication stream given a higher priority with `cudaStreamCreateWithPriority`. Because there are now multiple streams within the same work unit, CUDA events must be used to enforce dependencies between streams. It should be noted that such a simple bisection of streams may not be enough to obtain the desired degree of computation-communication overlap, as will be discussed in Section IV-B2.

B. *Support for Asynchronous Progress in the Runtime*

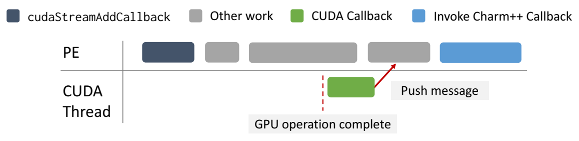
The application often needs to be notified when a GPU operation completes in order to invoke subsequent operations, to send a message after a H2D transfer, for example. The simplest method is performing a synchronization call such as `cudaStreamSynchronize` so that the host code blocks until the enqueued operations are complete. With overdecomposition, however, this hinders computation-communication overlap as the Charm++ scheduler will be blocked from making forward progress on communication and performing other chares' work.

A potential solution is using `cudaStreamAddCallback` (or more recently `cudaLaunchHostFunc`), which is an asynchronous CUDA API that enqueues a host function to be called once all currently enqueued work in the given stream completes. The problem is that it cannot be used directly by the programmer in a Charm++ application, as the designated function is executed on a separate thread created by the CUDA runtime which is dissociated from the Charm++ scheduler. Manually polling completion with CUDA events is also infeasible due to the scheduler-driven execution in Charm++.

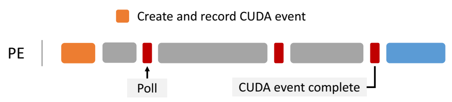
To support asynchronous progress of GPU operations in such scenarios, we implement two compile-time configurable mechanisms in the Charm++ runtime system: *callback-based* and *polling-based*. Either mechanism is exposed to the user via the following *Hybrid API (HAPI)* call:

```
// CkCallback is a Charm++ callback object
void hapiAddCallback(cudaStream_t stream,
    CkCallback* callback)
```

.



(a) Callback-based mechanism.



(b) Polling-based mechanism.

Fig. 4. Designs of asynchronous progress support for GPU operations in the Charm++ runtime system.

After GPU operations are enqueued in a CUDA stream, `hapiAddCallback` can be used to schedule a Charm++ callback (most often a chare's entry method) when they are complete. A message will be sent to the target chare once the Charm++ runtime detects all previous operations in the provided CUDA stream are complete, allowing the application to resume its work. The designs of the two different mechanisms underlying the API are discussed in the following.

1) Callback-based: The callback-based mechanism utilizes the CUDA callback feature to execute a codelet once the GPU operations in the specified stream complete. This codelet pushes a message containing the Charm++ callback object to the message queue of the PE which originally invoked `hapiAddCallback`. When the scheduler picks up the message, it invokes the Charm++ callback to execute the designated entry method on the target chare. Figure 4a illustrates this process. This mechanism essentially returns control back to the Charm++ scheduler from the CUDA-generated thread, in order to invoke the user-specified Charm++ callback.

2) Polling-based: The polling-based mechanism makes use of CUDA events to track the progress of GPU operations. When the user calls `hapiAddCallback`, a CUDA event is created and recorded. A HAPI event which encapsulates the CUDA event along with information about the user-specified Charm++ callback is also created. The HAPI event is then pushed to a FIFO event queue maintained by each PE, which is checked every time the scheduler picks up a message. Charm++ callbacks associated with the completed events in the queue are invoked before executing the next message. Figure 4b outlines this mechanism.

IV. Experimental Setup

Next, we describe the set of platforms and proxy applications used to evaluate the performance impact of our approach.

A. Platforms Used for Experiments

Two leadership-class GPU-accelerated supercomputers are used for performance evaluations: Summit at Oak Ridge National Laboratory and Lassen at Lawrence Livermore National Laboratory (a non-classified Sierra-like system). A brief summary of the hardware and software of these systems are provided in Table I. Note that Lassen has a limit of 256 nodes (1,024 GPUs) for regular jobs. The main architectural differences are the number of GPUs per node, intra-node GPU interconnect, and inter-node network topology. A more detailed comparison of these systems can be found in [6]. Because the two systems employ the same type of GPU, we expect the difference in performance to be derived largely from communication.

The same number of PEs as the total number of GPUs are used in the execution of Charm++ programs, with each PE assigned to one CPU core. On a single node of Summit, for example, six PEs (CPU cores) are mapped to six GPUs, with one PE per GPU.

B. Benchmarks

1) Jacobi3D: Jacobi3D is a simple Charm++ proxy application that performs the Jacobi iterative method on the GPU in a three-dimensional domain. The global grid is decomposed into cuboids, each contained within a chare. For the purposes of this work, Jacobi3D is configured to run a fixed number of iterations without convergence checks. Each Jacobi iteration consists of the following steps:

1) Perform Jacobi update on GPU (Equation 1)
2) For each halo to be sent to a neighbor,
    a) Invoke packing kernel to move halo data to contiguous buffer if necessary
    b) Device-to-host (D2H) transfer of halo buffer
3) Non-blocking exchange of halo data with neighbors
4) On receiving a message from a neighbor,
    a) Host-to-device (H2D) transfer of halo buffer
    b) Invoke unpacking kernel to move halo data into non-contiguous memory if necessary

The Jacobi update is a 3D stencil computation of the following:

$$A_{i,j,k} = \frac{1}{7}\times(A_{i,j,k}+A_{i-1,j,k}+A_{i+1,j,k}+A_{i,j-1,k}+A_{i,j+1,k} + A_{i,j,k-1} + A_{i,j,k+1}) \quad (1)$$

where $A_{i,j,k}$ is the block at position (i,j,k) of the global grid.

Each chare maintains separate compute and higher-priority communication CUDA streams as discussed in Section III-A. Packing/unpacking kernels and transfers between host and device are enqueued in the communication stream, whereas the Jacobi update kernel is offloaded in the compute stream.

2) MiniMD: MiniMD [7] is a proxy application for molecular dynamics simulation of a Lennard-Jones or EAM system, designed to be representative of the performance of the widely used LAMMPS [8] package. In this work, we employ a Lennard-Jones system without re-neighboring and Newton's third law for ghost atoms. MPI and Kokkos [9] performance

TABLE I
SUMMARY OF THE EXPERIMENTAL PLATFORMS

Platform	No. of nodes	CPU	GPU	GPUs/node	Network	MPI	CUDA
Summit	4,608	IBM Power9	NVIDIA Tesla V100	6	Mellanox EDR	IBM Spectrum	10.1
Lassen	792	IBM Power9	NVIDIA Tesla V100	4	Mellanox EDR tapered	IBM Spectrum	10.1

portability framework are used for execution on distributed GPU systems, where CUDA-aware MPI handles inter-GPU communication and Kokkos manages GPU execution through its CUDA backend.

To enable overdecomposition, we convert an MPI process to a chare array element and port the MPI communication routines to Charm++. Kokkos is retained as the performance portability layer for GPU execution, but several significant modifications are made to enable asynchronous progress. These include modifying Kokkos deep copies and parallel loops to use CUDA execution instances [10] associated with CUDA streams, and forced asynchronous kernel launches [11] to disable unwanted synchronization behaviors. It is important to note that the near-neighbor communication is modified from a set of `MPI_Sendrecv` calls to non-blocking communication routines in Charm++, in order to maximize overlap of computation and communication between different chares mapped to the same GPU. This trades off memory usage (as a separate set of send and receive buffers is needed for each neighbor exchange) for improvement in communication performance and more potential for computation-communication overlap.

Because of the current lack of support for direct GPU-GPU transfers in Charm++, CUDA-aware MPI calls are converted to explicit host-device transfers and host-to-host messages. However, the ability to hide the communication latency with overdecomposition allows the Charm++ version to outperform even the CUDA-aware MPI version, as discussed in Section V

The following main iteration loop is executed by each chare in the Charm++-Kokkos version of MiniMD:

1) Initial integration
2) Exchange of atom information
    a) Packing kernels
    b) Device-to-host (D2H) transfers
    c) Neighbor exchanges via host-to-host messages
    d) Host-to-device (H2D) transfers
    e) Unpacking kernels
3) Lennard-Jones force calculation
4) Final integration

Our first attempt at integrating CUDA streams in MiniMD involved using two streams per chare as in Jacobi3D, but it did not yield satisfactory performance due to the lack of computation-communication overlap. The issue was that the communication-related GPU operations (Steps 2a, 2b, 2d and 2e) were not prioritized as expected. Many of these operations were held back by force calculation kernels (Step 3) from other chares. Nevertheless, this is not an erroneous behavior, as CUDA stream priorities are merely hints to the CUDA scheduler and does not guarantee preemption of lower priority work in favor of higher priority work. In such situations, we need a more sophisticated design of CUDA streams interlaced with CUDA events to enforce inter-stream dependencies.

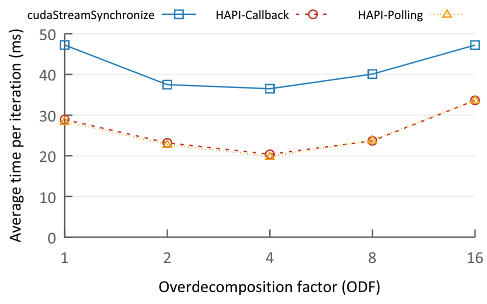


Fig. 5. Performance of Jacobi3D with varying overdecomposition factors on a single node of OLCF Summit.

Our design utilizes a total of five streams per *GPU* (instead of two streams per *chare*): one stream each for computational kernels (Steps 1, 3 and 4), packing kernels, D2H transfers, H2D transfers, and unpacking kernels. All streams aside from the compute stream are given higher priority. This allows communication-related operations to be properly prioritized and also overlap packing/unpacking kernels with D2H/H2D transfers. A potential drawback to this design is that compute kernels enqueued from different chares cannot execute concurrently, since there is only one compute stream per GPU. This can be fixed with a more complicated design with multiple compute streams, but it is left as future work. CUDA events are used to asynchronously enforce the following dependencies between streams: compute $\rightarrow$ packing, packing $\rightarrow$ D2H transfer, H2D transfer $\rightarrow$ unpacking, and unpacking $\rightarrow$ compute. With this design, we are able to effectively overlap computational kernels with host-device data transfers and host-to-host communication.

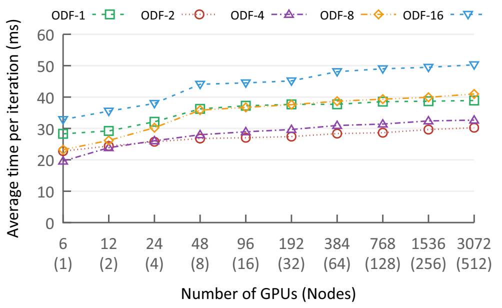
In both Jacobi3D and MiniMD, asynchronous progress of GPU operations is supported by the Charm++ runtime system through HAPI.

V. Performance Evaluation

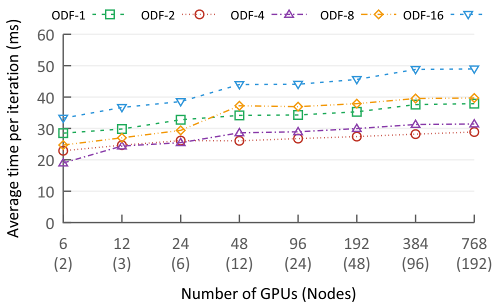
We evaluate the performance of our approach using two proxy applications, Jacobi3D and MiniMD, on two different GPU-accelerated platforms, Summit and Lassen. Performance is averaged across nine different measurements: three jobs each performing three runs of the same configuration.

A. Jacobi3D

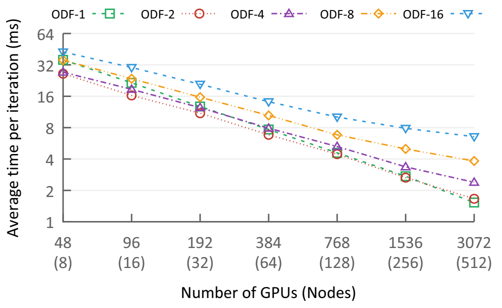
1) Single-node: We first evaluate the performance of Jacobi3D on a single node of Summit, with a global grid



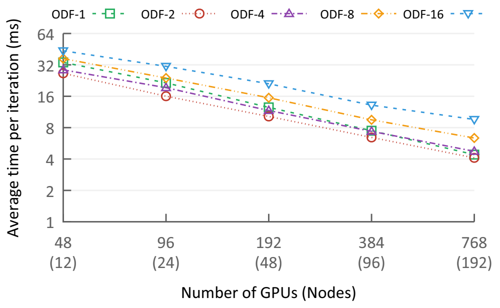
(a) Weak scaling on Summit.



(b) Weak scaling on Lassen.



(c) Strong scaling on Summit.



(d) Strong scaling on Lassen.

Fig. 6. Weak & strong scaling performance of Jacobi3D.

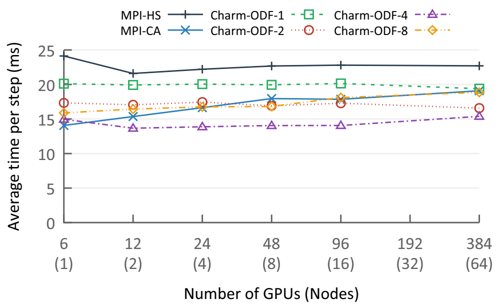
of 1,536 × 1,536 × 1,536. Figure 5 compares different mechanisms used to ensure halo data have been moved to host memory before performing neighbor exchanges: calling `cudaStreamSynchronize` on the communication stream which is the simplest approach, and using the Charm++ runtime support (callback-based and polling-based HAPI) to asynchronously invoke a Charm++ callback function once the operations in the communication stream are complete. The overdecomposition factor (ODF) is varied from one chare per GPU (MPI-like decomposition) to 16 chares per GPU; we expect overdecomposition to provide performance improvements due to computation-communication overlap up to a certain point, after which overheads from the finer granularity start to dominate performance.

`cudaStreamSynchronize` yields significant slowdowns over the versions with runtime support as PEs are fully blocked until all operations in the communication stream complete. Hence the Charm++ scheduler can neither initiate GPU operations of other chares nor progress host-side communication including the handling of incoming messages. The two HAPI mechanisms demonstrate up to 83% increased performance, with ODF-4 providing the largest performance improvement over ODF-1 of 43%. As expected, as the overdecomposition factor grows further, the increase in overall communication volume and overheads caused by smaller work units start to degrade performance.

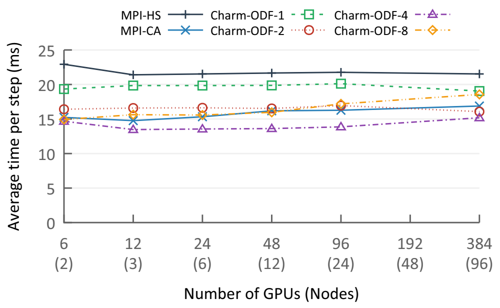
Although the callback-based mechanism performs similarly to the polling-based mechanism in Jacobi3D, it degrades performance in more fine-grained applications due to the CUDA-generated thread sharing a physical core with a Charm++ PE. We therefore adopt the polling-based mechanism for the following scaling studies.

2) *Weak scaling:* We perform weak scaling of Jacobi3D with a base problem size of 1,536 × 1,536 × 1,536. Each dimension of the grid increases twofold as the number of GPUs double, in x, y, z order. As shown in Figures 6a and 6b, ODF-4 performs the best until 12 GPUs on Summit and 24 GPUs on Lassen, obtaining up to 44% and 50% performance improvement over ODF-1, respectively. On larger node counts, however, ODF-2 begins to outperform ODF-4 with performance improvements compared to ODF-1 ranging between 24%-37% on Summit and 28%-33% on Lassen. We were unable to determine the exact cause of this crossover behavior and only observed longer idle times between iterations with ODF-4 after the crossover point. Nevertheless, an adequate degree of overdecomposition significantly improves performance by achieving computation-communication overlap.

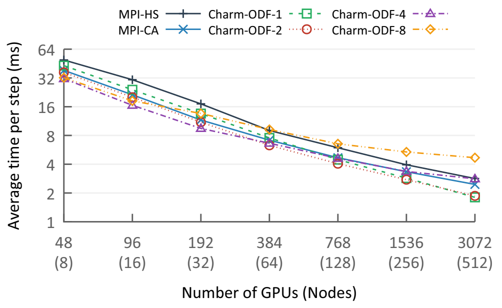
3) *Strong scaling:* Jacobi3D is strong scaled with a problem size of 3,072 × 3,072 × 3,072, from 48 GPUs to 3,072 and 768 GPUs on Summit and Lassen, respectively. As in Figures 6c and 6d, ODF-2 provides the best performance until 1,536 GPUs on Summit and 768 GPUs on Lassen, but its performance improvement over ODF-1 decreases from 35% to 3% on Summit and 27% to 7% on Lassen. With 3,072 GPUs on Summit, overdecomposition degrades performance as observed by the 8% slowdown with ODF-2. This is within our expectations, however, as the performance improvement achievable with overdecomposition diminishes as the size of



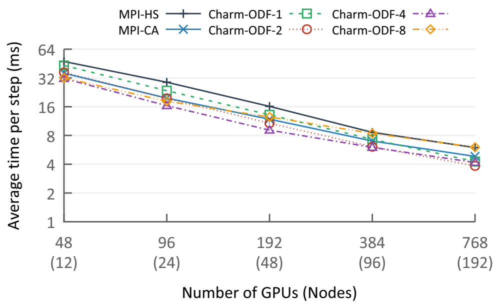
(a) Weak scaling on Summit.



(b) Weak scaling on Lassen.



(c) Strong scaling on Summit.



(d) Strong scaling on Lassen.

Fig. 7. Weak & strong scaling performance of MiniMD.

each work unit decreases with strong scaling. At large node counts, overdecomposition results in a small work unit being split up into even smaller pieces, aggravating fine-grained overheads.

B. MiniMD

The original CUDA-aware MPI version (marked MPI-CA) and modified host-staged version (marked MPI-HS, uses explicit copies between host and device) of MiniMD are benchmarked alongside the Charm++ versions employing overdecomposition. It should be noted that their performance is provided only for reference, since the Charm++ versions exercise a different communication pattern to facilitate computation-communication overlap, as described in Section IV-B2.

1) Weak scaling: We perform weak scaling of MiniMD with a base domain size of 192 × 192 × 192, which results in 28 million atoms that are split across 6 GPUs. As the number of GPUs double, each dimension of the grid is doubled, in x, y, z order. Figures 7a and 7b show the weak scaling performance up to 384 GPUs with domain size of 768 × 768 × 768 and atom count of 1.8 billion. We do not obtain results from 768 GPUs and onwards as an integer overflow occurs in the number of atoms. Results with 192 GPUs are not plotted as a NaN error causes computational kernels to run abnormally fast. These errors have been reported to the MiniMD developers.

It can be observed that the Charm++ version of MiniMD with an overdecomposition factor of four (Charm-ODF-4) performs the best except on a single node of Summit, where the CUDA-aware MPI version (MPI-CA) performs better. ODF-4 achieves speedups over ODF-1 ranging 26%-45% on Summit and 25%-47% on Lassen. Despite the lack of direct GPU-GPU transfers in the Charm++ versions, overlap of computation and communication achieved from overdecomposition allows Charm-ODF-4 to outperform MPI-CA in most configurations.[1]

2) Strong scaling: MiniMD is strong scaled with a domain size of 512 × 512 × 512 that contains 536 million atoms, from 48 GPUs to 3,072 GPUs on Summit and 768 GPUs on Lassen. As shown in Figures 7c and 7d, ODF-4 performs the best with performance improvements over ODF-1 between 36%-42% until 192 GPUs on Summit and 21%-44% until 384 GPUs on Lassen. Afterwards, ODF-2 provides the best performance with improvements decreasing from 19% to 3% on Summit and 11% on Lassen, except 3,072 GPUs on Summit where ODF-1 outperforms ODF-2 by 3%. Again, the results align with our expectation that the performance improvement obtainable with overdecomposition diminishes at the tail end of strong scaling, due to the smaller size of work units.

VI. Related Work

There has been extensive research on optimizing performance with computation-communication overlap. Task-based runtime systems including HPX [12], OmpSs [13], Legion [14], and StarPU [15] exploit overlap of computation and communication through different mechanisms, most of which support execution on GPU-accelerated systems. In particular, techniques to optimize computation-communication overlap

[1]The difference in communication pattern should also be taken into account.

by addressing the inefficient interactions between OmpSs and MPI are discussed in [5]. With a focus on overdecomposition and GPU execution, our work presents application design considerations and implementation details of a runtime feature for asynchronous progress, which can also be utilized by other task-based runtime systems and applications seeking to maximize computation-communication overlap on modern GPU systems.

VII. Conclusion

We discussed important considerations for achieving computation-communication overlap with overdecomposition on GPU systems, including the need to prioritize communication in the application and avoid synchronization with support from the runtime system. Techniques to address these issues have been presented and implemented in proxy applications and the runtime of the Charm++ parallel programming system. We demonstrated significant improvements in weak scaling performance of proxy applications on today's leadership-class GPU systems, albeit diminishing but expected returns with strong scaling.

This work marks an important milestone in our GPU roadmap, which includes on-going development of features such as integration of direct GPU-GPU transfers and dynamic load balancing with GPU loads. With improvements to GPU support in the Charm++ runtime system, we aim to make Charm++ a more attractive parallel programming model of choice for current and upcoming GPU-accelerated platforms.

Acknowledgment

We thank the Kokkos developer team at Sandia National Laboratory for providing assistance with technical issues.

This work was performed under the auspices of the U.S. Department of Energy (DOE) by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-814558).

This research was supported by the Exascale Computing Project (17-SC- 20-SC), a collaborative effort of the U.S. DOE Office of Science and the National Nuclear Security Administration.

This research used resources of the Oak Ridge Leadership Computing Facility at the Oak Ridge National Laboratory, which is supported by the Office of Science of the U.S. DOE under Contract No. DE-AC05-00OR22725.

References

[1] (2020) Summit user guide - system overview. [Online]. Available: https://docs.olcf.ornl.gov/systems/summit_user_guide.html#system-overview

[2] T. Papatheodore. (2018) Overview of high performance computing resources at the oak ridge leadership computing facility (olcf). [Online]. Available: https://www.olcf.ornl.gov/wp-content/uploads/2018/06/Intro_to_HPC_OLCF.pdf

[3] R. L. Graham, D. Bureddy, P. Lui, H. Rosenstock, G. Shainer, G. Bloch, D. Goldenerg, M. Dubman, S. Kotchubievsky, V. Koushnir, L. Levi, A. Margolin, T. Ronen, A. Shpiner, O. Wertheim, and E. Zahavi, "Scalable hierarchical aggregation protocol (sharp): A hardware architecture for efficient data reduction," in *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, 2016, pp. 1–10.

[4] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel programming with migratable objects: Charm++ in practice," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. IEEE Press, 2014, p. 647–658. [Online]. Available: https://doi.org/10.1109/SC.2014.58

[5] E. Castillo, N. Jain, M. Casas, M. Moreto, M. Schulz, R. Beivide, M. Valero, and A. Bhatele, "Optimizing computation-communication overlap in asynchronous task-based programs," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 380–391. [Online]. Available: https://doi.org/10.1145/3330345.3330379

[6] C. Zimmer, S. Atchley, R. Pankajakshan, B. E. Smith, I. Karlin, M. L. Leininger, A. Bertsch, B. S. Ryujin, J. Burmark, A. Walker-Loud, M. A. Clark, and O. Pearce, "An evaluation of the coral interconnects," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356166

[7] (2020) Mantevo/minimd. [Online]. Available: https://github.com/Mantevo/miniMD

[8] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1 – 19, 1995. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S002199918571039X

[9] H. Carter Edwards, C. R. Trott, and D. Sunderland, "Kokkos," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, p. 3202–3216, Dec. 2014. [Online]. Available: https://doi.org/10.1016/j.jpdc.2014.07.003

[10] (2020) Kokkos lectures module 5: Simd, streams and tasking. [Online]. Available: https://github.com/kokkos/kokkos-tutorials/blob/main/LectureSeries/KokkosTutorial_05_SIMDStreamsTasking.pdf

[11] (2020) Kokkos github issue #2545: Undesired fence-like behavior without calling a fence. [Online]. Available: https://github.com/kokkos/kokkos/issues/2545#issuecomment-555143767

[12] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2676870.2676883

[13] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: a proposal for programming heterogeneous multi-core architectures." *Parallel Processing Letters*, vol. 21, pp. 173–193, 06 2011.

[14] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[15] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," in *Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 863–874.

Appendix A
Artifact Description Appendix: Achieving Computation-Communication Overlap with Overdecomposition on GPU Systems

A. Abstract

This artifact description contains the necessary information to reproduce the experimental results presented in the paper, including installation and performance evaluation.

B. Description

1) Check-list (artifact meta information): Fill in whatever is applicable with some informal keywords and remove the rest

- **Algorithm:** Jacobi iteration, Lennard-Jones molecular dynamics
- **Program:** C++, MPI, Charm++, Kokkos
- **Compilation:** IBM XL C/C++ V16.1.1, NVCC 10.1.243
- **Hardware:** OLCF Summit, LLNL Lassen
- **Output:** Configuration, init time, total time, average iteration time for Jacobi3D; configuration, T/U/P values, performance summary for MPI & Kokkos version of MiniMD; configuration, T/U/P values, total time, average iteration time for Charm++ & Kokkos version of MiniMD
- **Experiment workflow:** Clone and build Charm++, clone and build Kokkos, compile Jacobi3D and MiniMD, execute binaries and observe results
- **Experiment customization:** Number of chares, grid dimensions, number of warmup iterations, number of iterations for Jacobi3D; number of MPI processes and standard parameters for MPI & Kokkos version of MiniMD; number of chares and standard parameters for Charm++ & Kokkos version of MiniMD
- **Publicly available?:** Yes

2) How software can be obtained (if available): All used software can be obtained from GitHub. Kokkos: https://github.com/kokkos/kokkos, Charm++: https://github.com/UIUC-PPL/charm, MiniMD (fork): https://github.com/minitu/miniMD Please refer to Section A-C for specific branches or tags.

3) Hardware dependencies: OLCF Summit and LLNL Lassen to replicate the results as is. Other systems with NVIDIA GPUs can also be used, but the results may differ.

4) Software dependencies: The following modules were loaded on the test systems: [Summit] `xl/16.1.1-5, spectrum-mpi/10.3.1.2-20200121, cuda/10.1.243`, [Lassen] `xl/2020.06.25, spectrum-mpi/rolling-release, cuda/10.1.243`

MPI, Kokkos, and Charm++ are required, see Section A-C for installation details.

C. Installation

1) Clone Kokkos version 3.2.

```
$ git clone -b 3.2.00 git@github.com:kokkos/
    kokkos.git
```

2) Build and install Kokkos (assumes Kokkos was cloned in `$HOME`).

```
$ cd kokkos && mkdir build && mkdir install &&
    cd build
$ cmake ../ -DCMAKE_CXX_COMPILER=$HOME/kokkos/
    bin/nvcc_wrapper -DCMAKE_INSTALL_PREFIX=
    $HOME/kokkos/install -DKokkos_ENABLE_CUDA=
    On -DKokkos_ENABLE_SERIAL=On -
    DKokkos_ARCH_POWER9=On -
    DKokkos_ARCH_VOLTA70=On
$ make -j && make install
```

3) Clone Charm++ with branch `jchoi/espm2-2020`.

```
$ git clone -b jchoi/espm2-2020 git@github.com:
    UIUC-PPL/charm.git
```

4) Build Charm++ with NVIDIA GPU support and PAMILRTS as the machine layer. Note that HAPI-polling is used as the default option.

```
$ cd charm
$ ./buildold charm++ pamilrts-linux-ppc64le smp
     cuda -j -g --with-production
```

5) Compile Jacobi3D. To use `cudaStreamSynchronize` instead of asynchronous progress support with HAPI, add `-DCUDA_SYNC` to Makefile.

```
$ cd charm/examples/charm++/cuda/gpudirect/
    jacobi3d
$ make -j
```

6) Clone MiniMD fork (confirm that branch is `charm`).

```
$ git clone git@github.com:minitu/miniMD.git
```

7) Build MPI-Kokkos and Charm++-Kokkos versions of MiniMD. Makefiles should be updated to point to the correct Kokkos and Charm++ paths. Note that the MPI-Kokkos version is built with CUDA-aware MPI as default. To use the modified host-staged mechanism, add `-DCOMM_HOST_STAGE` to Makefile.

```
$ cd miniMD/kokkos
$ make -j
$ cd ../charm
$ make -j
```

D. Experiment workflow

Once the above installation steps are complete, we are now ready to evaluate the performance of the proxy applications, Jacobi3D and MiniMD. This can be done via job submission scripts for Summit and Lassen provided in `scripts/{summit,laseen}/benchmark.sh`, or directly with MPI launchers (e.g. `jsrun`). The main differences between the execution commands on Summit and Lasen are the `jsrun` arguments due to the different number of GPUs per node, and the pemap string for mapping Charm++ PEs to CPU cores. Sample commands are given in the following section.

E. Evaluation and expected result

1) Running experiments: Performance evaluation can be done with the provided scripts on Summit and Lassen. Sample `jsrun` commands are shown below:

Summit:

```
// Jacobi3D with ODF-4 on 12 GPUs (weak scaling)
// Number of chares = (number of GPUs) * ODF
// = 12 * 4 = 48
jsrun -n12 -a1 -c1 -g1 -K3 -r6 ./jacobi3d -c 48 -x
    3072 -y 1536 -z 1536 -w 10 -i 100 +ppn 1 +pemap
    L0,4,8,84,88,92
```

```
// MPI-Kokkos MiniMD on 12 GPUs (weak scaling)
jsrun -n12 -a1 -c1 -g1 -K3 -r6 -M "-gpu" ./miniMD
  -i ../inputs/in.lj.miniMD -gn 0 -nx 384 -ny 192
  -nz 192 -n 100
```

```
// Charm-Kokkos MiniMD with ODF-2 on 24 GPUs
// (strong scaling)
jsrun -n24 -a1 -c1 -g1 -K3 -r6 ./miniMD -c 48 -i
  ../inputs/in.lj.miniMD -gn 0 -nx 512 -ny 512 -nz
   512 -n 100 +ppn 1 +pemap L0,4,8,84,88,92
```

Lassen:

```
// Jacobi3D with ODF-4 on 12 GPUs (weak scaling)
jsrun -n12 -a1 -c1 -g1 -K2 -r4 ./jacobi3d -c 48 -x
   3072 -y 1536 -z 1536 -w 10 -i 100 +ppn 1 +pemap
   L0,4,80,84
```

```
// MPI-Kokkos MiniMD on 12 GPUs (weak scaling)
jsrun -n12 -a1 -c1 -g1 -K2 -r4 -M "-gpu" ./miniMD
  -i ../inputs/in.lj.miniMD -gn 0 -nx 384 -ny 192
  -nz 192 -n 100
```

```
// Charm-Kokkos MiniMD with ODF-2 on 24 GPUs
// (strong scaling)
jsrun -n24 -a1 -c1 -g1 -K2 -r4 ./miniMD -c 48 -i
  ../inputs/in.lj.miniMD -gn 0 -nx 512 -ny 512 -nz
   512 -n 100 +ppn 1 +pemap L0,4,80,84
```

2) *Obtaining performance results:* The output in `stdout` can be observed to determine the performance of an execution.

Jacobi3D:

```
...
[CUDA 3D Jacobi example]
Grid: 1536 x 1536 x 1536, Block: 384 x 384 x 256,
    Chares: 4 x 4 x 6, Iterations: 100, Warm-up: 10,
     Bulk-synchronous: 0, Zerocopy: 0, Print: 0

Init time: 2.938 s
Total time: 3.230 s
Average iteration time: 32303.160 us
[Partition 0][Node 0] End of program
```

MPI-Kokkos MiniMD:

```
...
# Performance Summary:
# MPI_proc OMP_threads nsteps natoms t_total t_force
     t_neigh t_comm t_other performance perf/thread
    grep_string t_extra
12 1 100 56623104 1.488910 0.057636 0.000000
    0.287554 1.143720 3802990025.582719
    316915835.465227 PERF_SUMMARY 0.000000
```

`t_total` can be divided by the number of iterations to obtain the average time per iteration.

Charm++-Kokkos MiniMD:

```
...
[Block] Total time (exclude 1st iteration): 1.573846
     s
[Block] Average time per iteration: 0.015897 s
[Main] Blocks complete
[Main] Kokkos finalized
[Partition 0][Node 0] End of program
```

F. Experiment customization

1) *Jacobi3D:* The customizable parameters for Jacobi3D are the following:

```
-c [number of chares] -x [grid width] -y [grid
    height] -z [grid depth] -w [number of warmup
    iterations] -i [number of iterations]
```

2) *MiniMD:* The standard MiniMD parameters (force style, system size, density, force cutoff, etc.) can be provided to the MPI-Kokkos and Charm++-Kokkos versions, aside from some limitations to the Charm++-Kokkos version discussed in the following section.

G. Notes

As discussed in the paper, weak scaling performance of MiniMD with 768 and more GPUs are not provided due to the integer overflow in the number of atoms, as well as the execution with 192 GPUs due to a NaN error. These errors have been reported to MiniMD developers on GitHub.

The Charm++-Kokkos version of MiniMD currently has some limitations compared to the MPI-Kokkos version. It does not support EAM forces and reneighboring, as the necessary conversions to use CUDA execution instances in Kokkos and non-blocking commnication routines in Charm++ have not been completed. It also does not support LAMMPS input files and writing the output to a YAML file. These limitations will be addressed before the Charm++-Kokkos version is requested to be integrated into the mainline Mantevo/miniMD GitHub repository.