Efficient and Portable Support for Overdecomposition on Distributed GPU Platforms

Aditya Bhosale*, Anant Jain†, Shourya Goel†, Ritvik Rao*, Peddoju Sateesh Kumar†, Laxmikant Kale*
* University of Illinois Urbana-Champaign, Urbana, IL, USA
Emails: {adityapb, rsrao2, kale}@illinois.edu
† Indian Institute of Technology, Roorkee, Roorkee, India
Emails: {anant_j, shourya_g, sateesh}@cs.iitr.ac.in

***Abstract*—Overdecomposition enables asynchronous many-task runtimes such as Charm++ to overlap communication with computation and to balance load dynamically. Its benefits are well established on CPUs, but GPUs achieve peak efficiency with large kernels and bulk transfers, raising the concern that fine-grained execution will incur prohibitive launch, synchronization, and communication overheads. This paper investigates whether overdecomposition can be sustained efficiently on GPUs, and portably across vendors. We find that at fine granularity, the dominant costs arise not on the device but from serialization on the host, in vendor-specific ways, and that they can be mitigated transparently by the runtime: by parallelizing kernel submission and completion detection across a hierarchy of processes and threads sharing each GPU, and by routing every message through device-to-device paths suited to its destination. We evaluate these techniques in a single-source Charm++/Kokkos environment that runs unchanged on NVIDIA and AMD systems. On Task Bench, they reduce the minimum effective task granularity from 26–37μs to 4μs, and in weak and strong scaling studies of three mini-applications, Jacobi2D, MiniMD, and LULESH, on up to 64 A40 and 32 MI250X GPUs, overdecomposed execution matches MPI performance across a range of overdecomposition factors.**

***Index Terms*—performance, portability, GPU, overdecomposition, Kokkos, Charm++**

I. Introduction

Modern HPC has entered an era of unprecedented architectural diversity. The leading systems on the TOP500 list are distributed across at least three GPU vendors: Frontier and El Capitan are powered by AMD accelerators, Aurora by Intel, and Jupiter Booster, Eagle, and Alps by NVIDIA GPUs. Each vendor ships its own native programming model, HIP, SYCL, and CUDA respectively, and each uses a distinct network stack. For application developers and runtime system designers, this diversity has turned portability from a convenience to a first-class requirement.

Real applications increasingly exhibit dynamic behavior, for example, adaptive mesh refinement, multi-physics simulations, and irregular algorithms whose loads are data dependent and hard to statically predict. Hardware contributes another source of imbalance, from performance variability between identical GPUs, to thermal throttling, and network contention. The bulk-synchronous model that dominates large-scale GPU codes offers no inherent mechanism for tolerating this dynamism. Asynchronous Many Task (AMT) runtimes such as Charm++ [1], [17], HPX [16], Legion [3], etc. address this gap by decoupling the application's expression of parallelism from its hardware mapping.

Charm++ [1] addresses adaptivity through overdecomposition, i.e., dividing the computation into many more migratable objects (chares) than processing elements (PEs) and letting the runtime schedule, overlap, and migrate them with minimal application intervention. These benefits are well established on CPUs and validated at scale in production codes such as NAMD [22] and ChaNGa [15]. Extending the model to GPUs, however, poses a fundamental challenge: GPUs achieve peak efficiency with large, high-occupancy kernels and bulk transfers, while overdecomposition deliberately partitions work into smaller chunks, raising the concern that reduced occupancy, launch overheads, and synchronization costs will together outweigh its benefits.

Previous work from Choi et al. [8] quantified these costs for GPU-aware Charm++ applications on NVIDIA Summit, using a single-PE-per-GPU configuration, and proposed application-level mitigations such as kernel fusion to coalesce fine-grained packing/unpacking kernels with the computation kernel, and CUDA Graphs to amortize kernel launch overhead for iterative applications. While these techniques are effective, they require the application developer to restructure code around the runtime system's granularity. Their study was also limited to a single GPU vendor and a single communication stack. Whether *overdecomposed* GPU execution can be supported *portably* and *efficiently* from a single source across multiple GPU vendors and communication stacks, whether its overhead can be managed transparently at the runtime layer instead of the application layer, and how it scales with the overdecomposition factor on modern accelerators remain open questions.

In this paper we take a step towards answering them. We extend Charm++ with a portable GPU communication layer built on the Lightweight Communication Interface (LCI) [25] that operates over both ibverbs and libfabric, paired with Kokkos [10] for execution portability. The result is a single-source Charm++/Kokkos environment in which an overdecomposed application runs unchanged on NVIDIA and AMD GPUs.

Within this environment we pursue two application-agnostic, runtime-level approaches to managing the cost of

fine-grained execution on GPUs. First, we integrate the established technique of non-blocking GPU streams into the runtime so that each chare's fine-grained packing/unpacking kernels overlap with the computation kernels of other chares mapped to the same device; while stream-based overlap is well known, applying it transparently at the runtime layer relieves the application developer of managing streams explicitly. Second, and as the primary technical contribution of this work, we generalize the runtime's execution model into a hierarchy in which multiple processes are mapped to each GPU and multiple worker threads are mapped to each process. Mapping multiple PEs to a shared GPU allows kernels to be submitted from these PEs in parallel, mitigating the launch-side serialization that a single-PE design incurs as the overdecomposition factor grows. Both approaches are implemented entirely in the runtime, requiring no application-level restructuring, and complement the application-level techniques explored in prior work. On top of this foundation, we conduct a detailed empirical study of overdecomposition cost, measuring how the application overhead scales with the overdecomposition factor across vendor platforms, and characterizing the trade-offs across execution model configurations.

This paper makes the following contributions:

1) A portable GPU runtime for Charm++ built on LCI and Kokkos, supporting NVIDIA and AMD GPUs over both ibverbs and libfabric from a single source.
2) A hierarchical execution model for Charm++ with multiple processes per GPU and multiple worker threads per process, enabling parallel kernel submission to a shared device to mitigate launch-side serialization under overdecomposition; together with transparent, runtime-managed use of non-blocking streams to overlap each chare's pack/unpack kernels with other chares' computation.
3) A detailed empirical study of overdecomposition cost on AMD and NVIDIA GPUs, attributing per-chare overhead to its sources, characterizing how it scales with the overdecomposition factor, and analyzing the trade-offs across execution model configurations.

II. Background and Related Work

A. Performance Portability Frameworks

Kokkos [10], RAJA [4], and SYCL [19] provide single-source abstractions that compile to vendor-native backends, allowing parallel kernels to target CPUs and GPUs from any of the major vendors without source modification. These frameworks have been adopted at scale across the DOE exascale machines. By design, they scope themselves to within-node execution and do not address inter-node communication or runtime adaptivity. Our work uses Kokkos for execution portability and complements it with a portable communication layer.

B. Asynchronous Many-Task Runtimes on GPUs

Several AMT runtimes have been extended to multi-vendor GPU execution. Uintah [13] has been ported to AMD, NVIDIA, and Intel GPUs through Kokkos backends, demonstrating single-source portability for an asynchronous many-task framework across all three GPU vendors. HPX [16] achieves portable GPU execution through its integration with Kokkos [9], exposing asynchronous kernel launches as futures that compose with its global task graph. PaRSEC [7] schedules tasks from a dataflow representation onto NVIDIA, AMD, and Intel GPUs, managing data movement and device queues in the runtime. StarPU [2] similarly supports multiple accelerator backends and uses performance-model-driven scheduling to place tasks across heterogeneous resources. These systems organize computation around explicit task graphs or dataflow dependencies. Charm++ instead schedules message-driven, migratable objects, which makes overdecomposition a central, tunable parameter of execution. The cost of sustaining that overdecomposition on GPUs, and the execution-model support needed to manage it portably across vendors and communication stacks, is the focus of this work.

C. Charm++ and Overdecomposition

In Charm++, the programmer decomposes the computation and its associated data into C++ objects called chares, organized into one or more multi-dimensional collections. The runtime system, rather than the programmer, assigns chares to PEs, and the application remains oblivious to their placement. A chare addresses another by collection name and index through an asynchronous method invocation, which returns immediately while the runtime delivers the message to the target chare's PE, where the scheduler instance eventually invokes the method. Because the chare locations are managed by the runtime, it can migrate chares between processors, enabling instrumentation-based dynamic load balancing. Overdecomposition, ie. creating many more chares than PEs, is essential to support dynamic load balancing.

Charm++ has been used to develop several production applications that exercise these capabilities at scale, including NAMD [22] for biomolecular simulation, ChaNGa [15] for cosmological N-body simulation, OpenAtom [14] for ab initio molecular dynamics, and SpECTRE [20] for relativistic astrophysics. Beyond load balancing, the same migratability that underlies these applications also supports a broader set of runtime services such as fault-tolerance [21] in which chares are recovered on surviving processors after a failure, malleability [11] in which a running job dynamically grows or shrinks its processor allocation enabling efficient use on cloud resources [5], [6].

III. Execution Model

A. Managing Overdecomposition

Overdecomposition implies that each processing element (PE) hosts many objects, or chares, each launching its own kernels. A natural question is how many chares are appropriate. In the traditional CPU-only setting, Charm++ applications typically perform best with roughly 8 to 16 chares per PE, where a PE corresponds to a core on which a Charm++ scheduler instance runs. This degree of overdecomposition

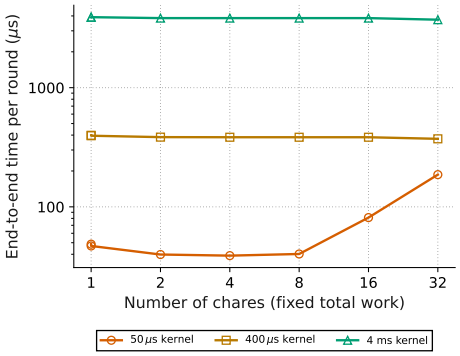
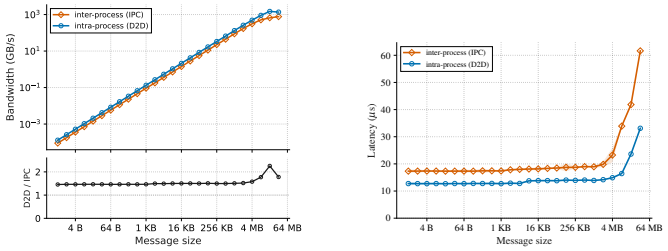


Fig. 1: Effect of overdecomposition on kernel runtimes



(a) Bandwidth (b) Latency

Fig. 2: Intra-process and inter-process communication bandwidth and latency between 2 PEs mapped to the same GPU

gives the load balancer sufficient flexibility to migrate chares toward or away from individual cores in order to restore balance. With fewer chares per PE, quantization effects begin to dominate: when work units are few and coarse, the load balancer cannot make fine-grained adjustments, and residual imbalance becomes unavoidable.

The introduction of GPUs raises an important question: is the core still the right unit of scheduling? With well over a hundred cores on a modern node, requiring 8–16 chares per PE would force an extremely fine-grained decomposition. The key observation is that in modern GPU-accelerated applications, the computational load resides almost entirely on the GPUs; host cores serve primarily to coordinate execution and to perform small, unavoidable computations. We therefore conclude that overdecomposition should be defined with respect to the GPU, i.e., 8–16 chares per GPU device. This reframes the objective of this paper as a more tractable one: can we sustain a grain size fine enough to yield 8–16 chares per GPU without undue overhead? What are the obstacles, and what techniques can be developed to overcome them?

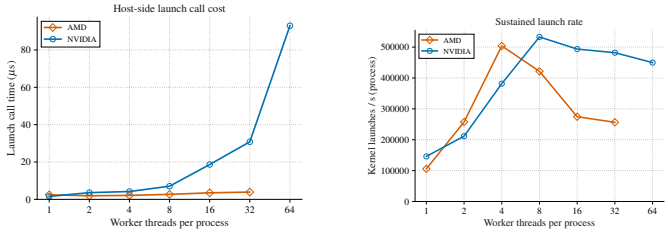
To study the effect of overdecomposition on kernel runtimes, we designed an experiment in which a kernel of fixed problem size is decomposed into a varying number of chares, all driven by a single PE and launched on a single GH200 GPU on TACC's Vista supercomputer. Each chare launches its portion of the work as a separate kernel, which we refer to as a sub-kernel, and, upon asynchronously detecting its completion, launches the next round's sub-kernel, for several rounds. Figure 1 shows the end-to-end time per round as the overdecomposition factor increases, for a range of kernel granularities; each curve is labeled by the duration of the undecomposed kernel. Overdecomposition proves to be remarkably cheap: the 400μs and 4ms kernels are unaffected at every factor we measure, and even the 50μs kernel benefits from overdecomposition, running faster than its undecomposed form through 8 chares, at which point each sub-kernel is only about 6μs, as one chare's sub-kernel execution overlaps the completion detection of the others. Only at very fine grains, sub-kernels of a few microseconds and below, does the fixed host-side cost of each sub-kernel (the launch call, completion detection, and scheduling) come to dominate and the runtime rise. Sustaining those decompositions requires spreading chares across multiple PEs per device, so that the host-side overheads are themselves parallelized.

Mapping a single process with many worker threads to each GPU minimizes communication cost: chares on different threads share one driver context, so a message between them reduces to a single device-to-device copy. The shared context, however, carries a host-side cost. Kernel submission within a context is serialized, so concurrent launches from multiple threads contend, and the time each thread spends queueing in the CUDA/HIP runtime grows with the thread count. Conversely, mapping multiple single-threaded processes to each GPU gives every process its own context and eliminates this contention, but messages between chares in different processes must then take the IPC path, which stages each transfer through an intermediate buffer with additional copies and synchronization, and is substantially more expensive than a direct device-to-device copy as can be seen in figure 2.

Both configurations are further constrained by limits of the GPU software stack that are invisible to the application. The multi-threaded configuration is bounded by its hardware queues: all streams of a process context are multiplexed onto a small number of per-process queues, and exceeding the hardware's capacity degrades silently, through stream aliasing on NVIDIA and queue time-slicing by the hardware scheduler on AMD. The multi-process configuration depends on MPS for concurrency on NVIDIA hardware, since kernels from separate contexts otherwise time-slice, and MPS brings device-wide budgets of its own: its clients draw hardware channels from a fixed shared pool and the number of client processes per device is capped. ROCm provides no MPS equivalent, so on AMD hardware the multi-process configuration time-slices regardless.

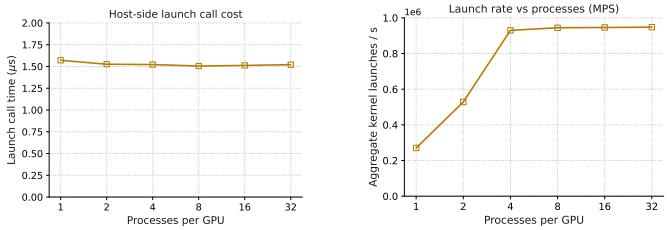
To study the effect of thread contention on kernel launch overhead, we construct an experiment in which a varying number of host threads concurrently submit empty kernels to a single GPU. Figure 3a shows the host-side launch overhead as a function of thread count on an NVIDIA GH200 and an AMD MI250X. On the GH200, the launch overhead rises sharply with the number of threads, indicating substantial contention in the launch path.

This contention, and not the device, is what limits the launch



(a) Kernel launch overhead (b) Kernel launch rate

Fig. 3: Kernel launch overhead and launch rate with varying number of threads

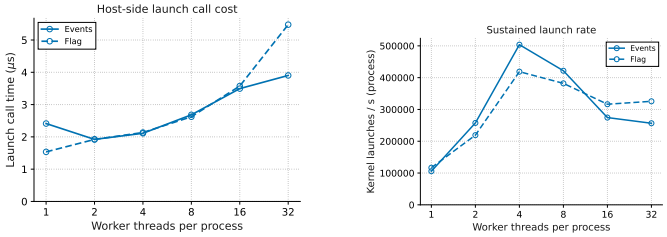


(a) Kernel launch overhead (b) Kernel launch rate

Fig. 4: Kernel launch overhead and launch rate with varying number of processes

Threads	MI250X (μs)	GH200 (μs)
1	10.2	0.163
2	20.3	0.189
4	40.6	0.228
8	81.2	0.229
16	154.4	0.229

TABLE I: Cost of querying an in-flight event with T threads polling concurrently, each on its own event. On MI250X queries serialize process-wide, growing linearly in T; on GH200 they are two to three orders of magnitude cheaper and nearly flat.



(a) Kernel launch overhead (b) Kernel launch rate

Fig. 5: Kernel launch overhead and launch rate with varying number of threads

rate of a multithreaded process. A single process peaks at 0.57M launches per second at 16 threads and declines with additional threads as the serialized submission path saturates. To measure what the GPU itself sustains, we repeat the experiment with single-threaded processes sharing the GPU under MPS: the aggregate rate reaches 0.95M launches per second and remains flat from 8 through 32 processes, with the per-process launch overhead constant at 1.5μs. A multithreaded process therefore attains only about 60% of the device's sustainable dispatch rate.

The launch overheads differ across platforms in a way that is initially counterintuitive: on GH200 the launch overhead grows from 1.5μs at 1 thread to 95μs at 64 threads, while on MI250X it stays below 4μs, despite MI250X delivering far fewer launches per second. Between launches, each thread repeatedly queries the event recorded behind its kernel (`cudaEventQuery` or `hipEventQuery`) to detect completion, so a cycle alternates between two shared resources: the submission path and the query path. Table I shows the query overhead with varying number of threads on GH200 and MI250X. On GH200 queries are cheap and concurrent, so the serialized submission path is the only place to wait and all contention appears in the launch call. On MI250X a query against a running kernel is significantly more expensive than a submission and serializes across threads, so threads wait there instead; launches arrive at the submission path sparsely and the call reads as uncontended. The low launch overhead on MI250X is therefore a result of starvation by the query path, not of cheaper launches.

The serialized query path is a property of the event API, not of completion detection itself. To avoid it, we implemented an alternative flag-based detection mechanism in the runtime. When a kernel is enqueued, our implementation enqueues behind it a stream-ordered write (`hipStreamWriteValue32`) of a sequence number into a per-thread slot of pinned host memory; the scheduler then detects completion with a plain load of that slot, which costs nanoseconds, acquires no lock, and is unaffected by the number of polling threads. Figure 5 repeats the launch-rate experiment on MI250X with both mechanisms. Event polling peaks at 0.5M launches per second at 4 threads and falls to 0.26M at 32 threads as the serialized queries consume the scheduler threads. Flag-based detection sustains 0.32M from 16 threads onward, and its launch-call time rises where event polling's stayed low, confirming that the waiting has migrated from the query path back to the submission path. Below 8 threads, event polling retains an advantage of at most 17%: a stream write is a slightly more expensive submission than an event record, and with few pollers the query path is uncontended, so only the costlier write is visible. We expose the detection mechanism as a runtime option. Flag-based detection is the appropriate choice where serialized queries dominate, which on MI250X occurs beyond 8 threads per process; at lower thread counts, and on platforms such as GH200 where queries are inexpensive and concurrent, event polling remains preferable.

These results indicate that sustaining overdecomposition at a fine granularity requires multiple PEs per device so that host-side kernel management overheads are incurred in parallel. Having these PEs as threads within a single process, however, serializes kernel submission and oversubscribes per-context

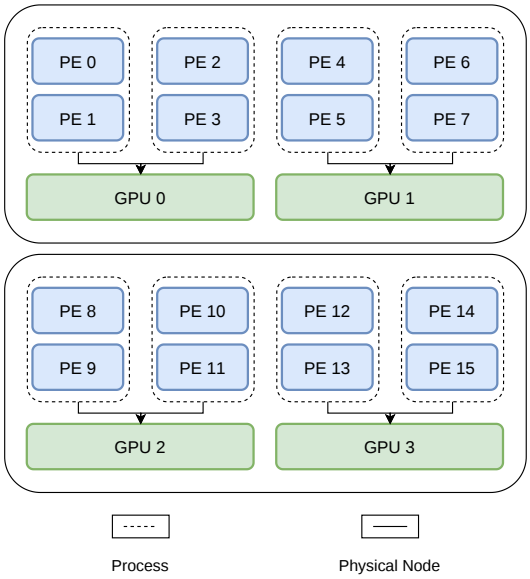


Fig. 6: Execution model for GPU-aware Charm++ applications

hardware queues, whereas distributing them across single threaded processes imposes IPC staging costs on all inter-PE communication and consumes device-wide MPS resources for which AMD GPUs offer no counterpart. These observations motivate an execution model in which multiple processes may be mapped to a GPU and multiple worker threads to a process, such that the process and thread counts can be selected at launch time without modification to the application code. The following section describes this hierarchical execution model.

B. Hierarchical Execution Model

We organize the runtime into a hierarchy of threads, processes, and GPUs, illustrated in Figure 6. Each GPU is shared by one or more processes, and each process contains one or more worker threads. Chares are scheduled onto PEs by the Charm++ runtime, and each chare issues its GPU kernels to the GPU associated with its process. PEs within the same process share a CUDA or HIP driver context, enabling direct device-to-device copies between chares. Cross-process GPU messages uses CUDA or HIP IPC on the same node and GPU-aware RDMA across nodes.

Wider processes, ie. more PEs per process and fewer processes per GPU, keeps more message traffic within a single driver context, where it benefits from direct device-to-device copies. Smaller processes routes more traffic through the IPC mechanism which requires two intermediate copies into pre-allocated buffers on both the sender and receiver. Wider processes, however, incur an overhead due to thread contention on the host for kernel launches.

To characterize the process/thread trade-off directly, we use a synthetic benchmark that holds every resource fixed except the position of the process boundaries. A single GPU is driven by 16 PEs organized into P processes of $16/P$ threads each, for $P \in \{1, 2, 4, 8, 16\}$, with multi-process configurations

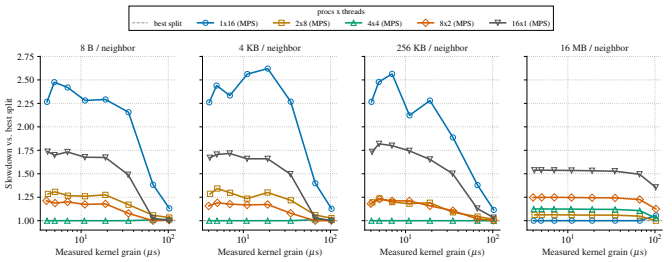


Fig. 7: Relative runtimes for different runtime configurations for varying kernel grain and message sizes

sharing the GPU under MPS. The application is a ring of 32 chares block-mapped to the PEs, so the fraction of neighbor messages that cross a process boundary rises with P while the number of chares, PEs, and their core placement stay identical. In every iteration each chare launches a kernel and exchanges a message with each ring neighbor, and the next iteration begins when both complete. We sweep the kernel grain from 1 to 100μs in factor-of-two steps, and the message size from 8B to 16MB. We report the mean iteration time of each configuration, taking the median over three independent runs.

Figure 7 reports the results as the slowdown of every split relative to the best split at that point for each grain and message size. A value of one marks the best achievable configuration. At fine grains and small messages the single wide process is the worst choice, up to $2.3\times$ the best split, because all 16 threads serialize on one submission path; distributing the same PEs across processes removes this contention. At the largest message sizes the ordering reverses: messages that cross a process boundary pay two additional copies through the device communication buffer, so the single process is best and the fully multi-process split trails by roughly $1.5\times$. Increasing the grain compresses all slowdowns toward one at every message size, as kernel execution hides the host-side costs.

Finally, we repeat the decomposition experiment of Figure 1 with the chares spread across PEs instead of driven by one, assigning each chare its own PE. We use single-threaded processes sharing the GPU under MPS since the benchmark involves no GPU communication. Figure 8 shows that the per-kernel host work that previously accumulated serially on one thread now proceeds concurrently across PEs, and the end-to-end time remains flat in the number of chares down to the finest kernel grains.

C. Using Kokkos in Charm++ runtime

Kokkos is designed primarily around a synchronous, MPI-style execution model, and integrating it with Charm++ requires care to preserve asynchrony and overlap. One source of overhead is implicit synchronization: operations such as temporary buffer deallocation and `Kokkos::resize` internally invoke `Kokkos::fence` due to Kokkos's reference-counted memory management, and in a multithreaded Charm++ ex-

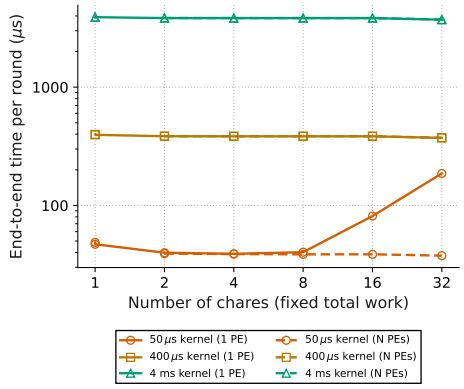


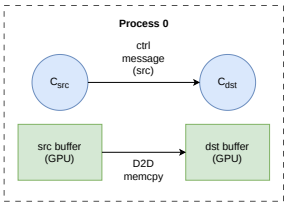
Fig. 8: Effect of overdecomposition on kernel runtimes with multiple PEs per GPU (N is the number of chares)

ecution these fences stall unrelated chares. We avoid them by making frequently used `Kokkos::Views` persistent class members and drawing scratch space from preallocated buffers rather than allocating temporaries during execution. A second source is Kokkos's functor caching, which adds synchronization across kernel launches; since our applications rarely launch identical kernels back to back, the cache provides little benefit, and we disable it using Kokkos's lightweight kernel hints.

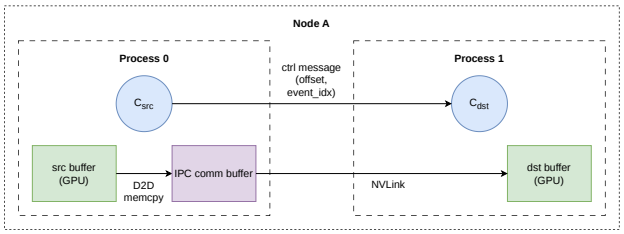
To run concurrently on multiple streams, we use separate Kokkos execution spaces, which introduces further pitfalls: `Kokkos::deep_copy` without an explicit execution space synchronizes on the host, and kernels launched without one run on the default stream, which implicitly synchronizes with all other streams. Because creating an execution space is itself expensive, involving device initialization and host-to-device copies, we create the spaces once at startup and reuse them for the lifetime of the application.

IV. Portable Communication Layer

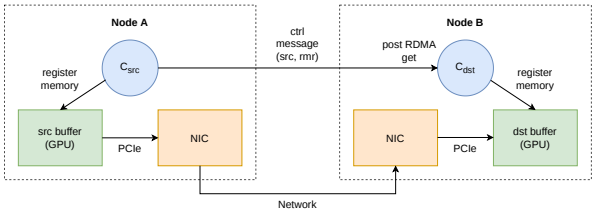
Overdecomposition of work and data units inherently fragments communication: the same total data volume is partitioned into smaller but more numerous messages. Consider a two-dimensional stencil computation as an illustrative example; overdecomposing by a factor of four halves the size of each halo-exchange message while doubling the message count along each dimension. The situation is further complicated by the heterogeneity of communication paths that arises under overdecomposition. Depending on chare placement, a given message may be destined for a chare on the same device and the same process, a chare on the same device but a different process, a chare on a peer device within the same physical node, or a chare on a remote node. Since these paths differ substantially in their performance, mitigating overdecomposition overheads requires that every message be routed through the transport mode best suited to its destination. The Charm++ runtime system enables this by resolving the location of the destination chare at send time, supported by a runtime option that pre-populates the location table,



(a) Intra-process GPU communication



(b) Intra-node, inter-process GPU communication



(c) Inter-node GPU communication

Fig. 9: GPU communication protocols in Charm++

thereby allowing the messaging layer to select the appropriate transport protocol. The following section describes each of these transport modes in detail.

A. Transport selection

The runtime's messaging is built on LCI [25], which provides asynchronous two-sided and one-sided operations over ibverbs and libfabric backends. Because LCI abstracts the network API, the same runtime code runs over InfiniBand and Slingshot. When a chare sends a device zerocopy message, the runtime selects one of three transport paths based on the location of the source and destination chares. The three transport paths are illustrated in Figure 9.

1) Intra-process transfers: When the source and destination chares reside in the same process, the transfer is a single asynchronous device-to-device copy, issued through the native CUDA/HIP APIs on the stream specified in the post entry method.

2) Intra-node, inter-process transfers: When the source and destination chares are in different processes on the same physical node, the transfer uses IPC. To avoid the high cost of opening an IPC memory handle for every message, we stage

transfers through a communication buffer: at startup, each process pre-allocates a shared device communication buffer and a pool of cross-process synchronization events, and maps every peer's buffer and events into its own address space, exchanging handles via POSIX shared memory. A transfer then involves two device-to-device copies, from the source buffer into the source's communication buffer and from there into the destination buffer, with an IPC event on the destination process ensuring the first copy completes before the second begins. The chare-level control message carries the offset into the source communication buffer and the index of the event used for synchronization.

3) Inter-node transfers: For chares on different nodes, we use one-sided RDMA. The source PE registers its buffer and sends a control message with the buffer address to the destination PE, which issues a one-sided get that pulls the data directly from the source GPU into its own.

V. Experimental Results

A. Task Bench

We evaluate the end-to-end effect of the execution model with Task Bench [23], [24], using a one-dimensional stencil task graph with a compute-bound kernel. Following the Task Bench methodology, we sweep the per-task granularity downward and report the minimum effective task granularity (METG), the smallest average task duration at which a configuration sustains at least 50% of the achievable compute rate. The results are shown in Figure 10.

All configurations reach the same 93% efficiency plateau at coarse grains, so METG isolates how much overhead each configuration adds per task. We run the NVIDIA experiments on NCSA Delta with 4 A40 GPUs per node. For Charm++ we launch one process per GPU and sweep the number of threads per process; the graph width fixes the overdecomposition. We compare against two MPI references, each running one rank per GPU: one synchronizes the stream on the host each timestep, the other enforces dependencies on the device through IPC CUDA events.

Without overdecomposition (width 4 on one node, one task per GPU per step), every system is confined to a synchronous launch-and-wait cycle, and METG is set by the full per-task round trip: 26 – 37μs across all three systems. Overdecomposition reduces this overhead even with a single PE. At width 32, eight chares per GPU driven by one PE reach an METG of 10.5μs, a 3.5$\times$ improvement, as the runtime pipelines one chare's task management overhead under another's kernel. Additional PEs then parallelize the remaining serial host work: METG falls to 6.1μs at 2 PEs and saturates at 4μs at 4 PEs (3.9μs at 8 PEs). At four nodes with width 128 (Figure 10c), the METG of each configuration is close to its single-node value, while MPI remains above 10μs.

B. Scaling Results

We evaluate the weak and strong scaling performance of overdecomposition using three mini-applications: Jacobi2D, MiniMD, and LULESH. For each, we compare our Charm++/Kokkos implementation against a corresponding MPI/Kokkos implementation across a range of overdecomposition factor (ODF) configurations. Experiments on NVIDIA hardware are conducted on the A40 compute nodes of NCSA Delta, where each node contains four NVIDIA A40 GPUs connected via PCIe Gen4, and nodes are interconnected by an HPE/Cray Slingshot 11 network. Experiments on AMD hardware are conducted on OLCF Frontier, where each node contains four AMD Instinct MI250X GPUs. Each MI250X comprises two Graphics Compute Dies (GCDs), so a Frontier node exposes eight addressable GPU devices. Frontier nodes are likewise interconnected via HPE Slingshot.

1) Jacobi2D: Jacobi2D is a stencil-based application that solves the Laplace equation on a two-dimensional grid. The grid is partitioned among chares (or MPI ranks in the reference implementation), which communicate via halo exchanges. Each experiment runs for 100 iterations without convergence checks.

a) Weak scaling on NVIDIA: Figure 11a presents weak scaling results for Jacobi2D with a base domain size of 32768×32768 per GPU. As the number of GPUs doubles, the grid dimensions are enlarged alternately along each axis to maintain a constant workload per GPU, scaling up to 32 GPUs across 8 nodes. Across the evaluated range, all ODF configurations closely match the MPI implementation, indicating that overdecomposition introduces little additional overhead while still enabling capabilities such as dynamic load balancing.

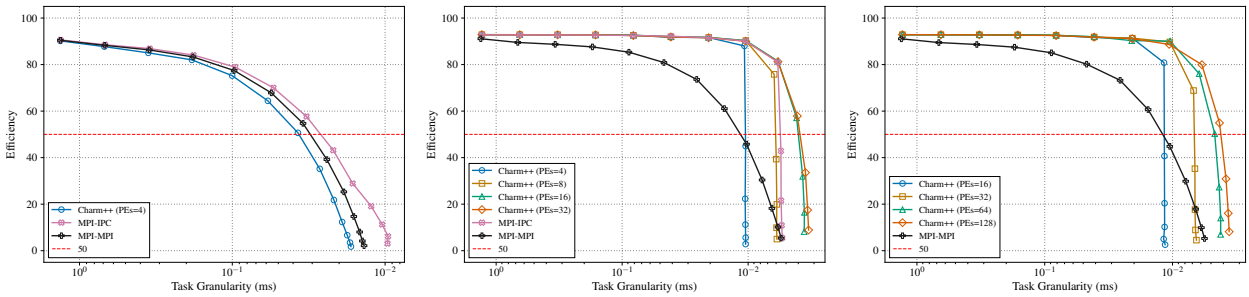
b) Strong scaling on NVIDIA: For the strong scaling experiments, we fix the domain size at 131072×98304 and scale from 8 GPUs (2 nodes) to 64 GPUs (16 nodes). As shown in Figure 11b, the ODF configurations continue to track MPI performance closely even at larger scales, consistent with the weak scaling observations.

c) Weak scaling on AMD: On the AMD system, weak scaling is evaluated using the same base domain size as the NVIDIA experiments, scaling from 1 GPU to 32 GPUs across 4 nodes. As shown in Figure 11c, the time per step increases only marginally with GPU count, mirroring the behavior observed on the A40 nodes. Notably, however, overdecomposition yields a clear reduction in execution time relative to both MPI and Charm++ without overdecomposition.

d) Strong scaling on AMD: Strong scaling on AMD uses the same fixed domain size as the NVIDIA experiments, scaling from 8 GPUs (1 node) to 32 GPUs (4 nodes).

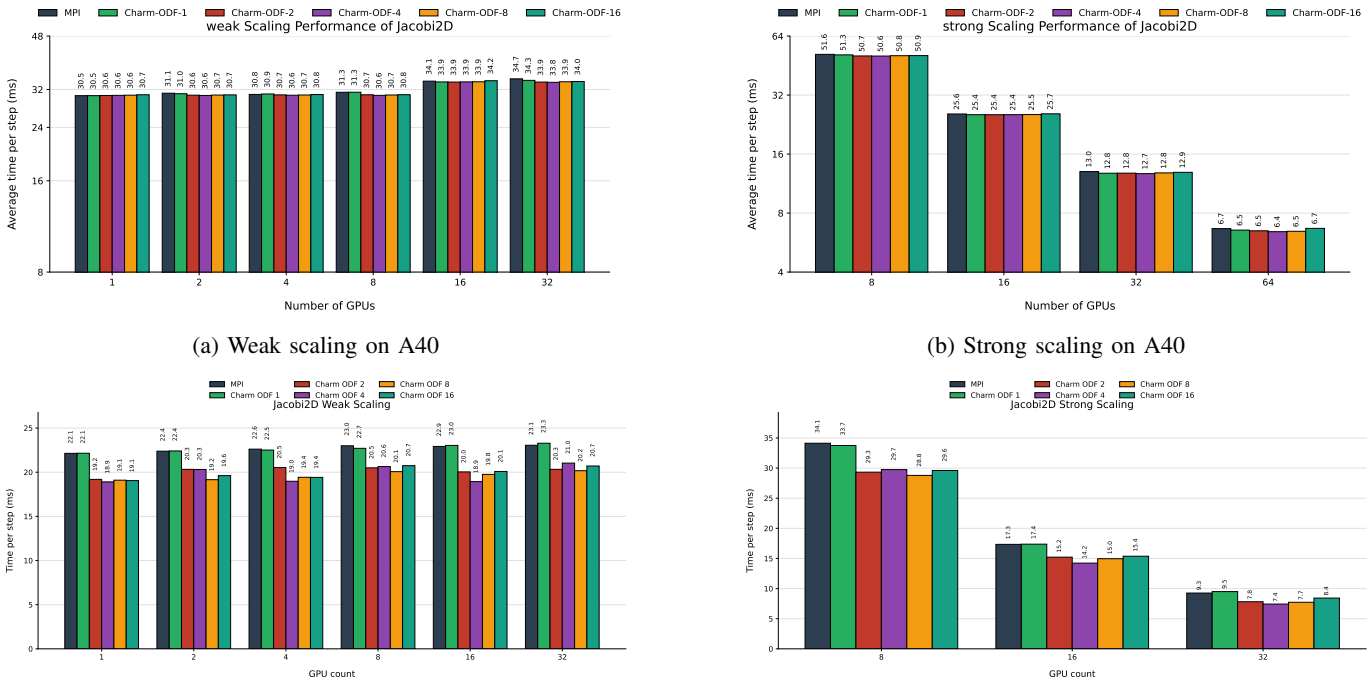
2) MiniMD: MiniMD [12] is a molecular dynamics proxy application that follows many of the same design principles as the production MD code LAMMPS. It employs a spatial decomposition in which each Charm++ chare or MPI rank, arranged in a three-dimensional grid, owns a subset of the simulation domain. The application exhibits a mixture of communication patterns arising from its distinct communication routines. We run MiniMD for 100 iterations, disabling reverse communication and safe-exchange checking.

a) Weak scaling on NVIDIA: We begin with a base domain size of $100 \times 100 \times 200$ and enlarge one dimension



(a) 1 node, 4 GPUs, width 4 (b) 1 node, 4 GPUs, width 32 (c) 4 nodes, 16 GPUs, width 128

Fig. 10: Task Bench 1D Stencil METG



(a) Weak scaling on A40 (b) Strong scaling on A40

(c) Weak scaling on MI250X (d) Strong scaling on MI250X

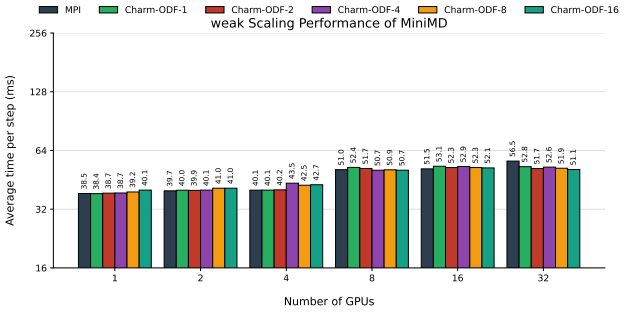
Fig. 11: Jacobi2D weak and strong scaling on A40 and MI250X GPUs.

alternately at each scaling step, scaling from 1 GPU to 32 GPUs across 8 nodes. For 1 to 16 GPUs, the overall trend resembles that of Jacobi2D: runtimes remain comparable between the MPI implementation and the Charm++ implementation across ODF configurations. However, as shown in Figure 12a, Charm++ begins to outperform MPI at larger scales. At 32 GPUs, all Charm++ configurations outperform MPI by 6.5–9.5%. Since ODF-1 shares MPI's decomposition, the baseline gap likely reflects differences between the LCI-based and MPI communication paths rather than overdecomposition; the further improvement from ODF-1 to the higher ODF configurations, roughly 3%, is attributable to communication–computation overlap.

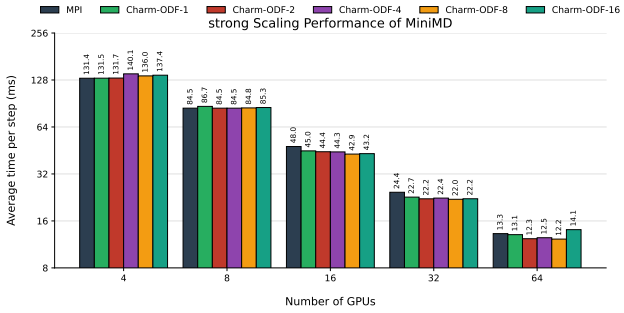
b) Strong scaling on NVIDIA: We perform strong scaling experiments with a fixed domain size of $300 \times 300 \times 300$, scaling from 4 GPUs (1 node) to 64 GPUs (16 nodes). As in the weak scaling study, the MPI and Charm++ implementations exhibit broadly comparable performance across the scaling range. At 32 and 64 GPUs, however, the ODF-16 configuration begins to perform slightly worse than intermediate ODF values. We attribute this to the small per-GPU problem sizes at these scales, where excessive overdecomposition introduces runtime overheads that can no longer be effectively hidden.

c) Weak scaling on AMD: Using the same base domain size as the NVIDIA experiments, we scale MiniMD from 1 GPU to 16 GPUs (2 nodes). As shown in Figure 12c, the Charm++ and MPI implementations show no discernible performance gap, consistent with the behavior observed on NVIDIA hardware.

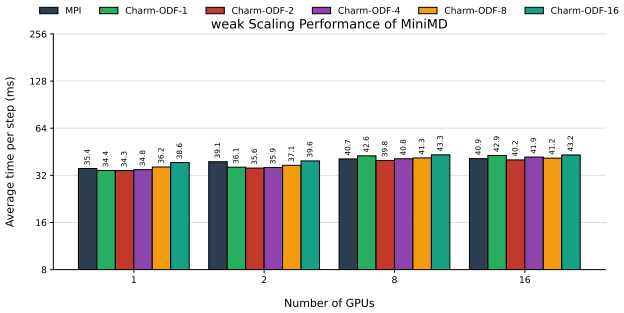
d) Strong scaling on AMD: We perform strong scaling experiments with the same fixed domain size as the NVIDIA runs, scaling from 4 GPUs (1 node) to 32 GPUs (4 nodes). The



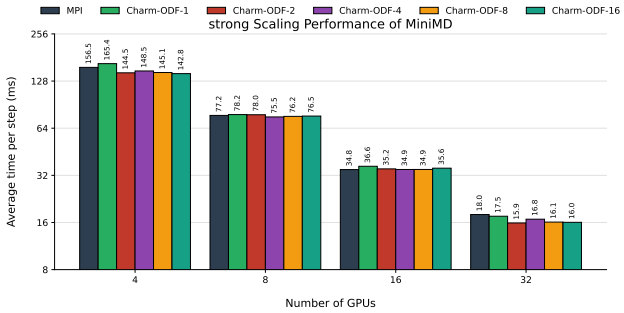
(a) Weak scaling on A40



(b) Strong scaling on A40



(c) Weak scaling on MI250X



(d) Strong scaling on MI250X

Fig. 12: MiniMD weak and strong scaling on A40 and MI250X GPUs.

results in Figure 12d indicate that Charm++ and MPI perform on par with one another, again aligning with the NVIDIA observations.

3) LULESH: LULESH [18] is a shock-hydrodynamics mini-application that solves the Sedov blast wave problem on a 3D hexahedral mesh. The domain is partitioned across Charm++ chares or MPI ranks, each of which owns a subdomain and exchanges state with its neighbors across boundary elements. The application can also model load imbalance across domains, making it a useful benchmark for future load-balancing studies. Both the MPI and Charm++ implementations require the number of domains to be a perfect cube.

This perfect-cube constraint often makes the achievable ODF fractional: a target ODF of 4 on 32 GPUs would require 128 chares, so we use the nearest perfect cube, 125, for an effective ODF of $125/32 \approx 3.91$. We run LULESH for 50 to 100 iterations depending on the scaling configuration.

The constraint also complicates weak scaling. The global mesh must have equal side lengths in all three dimensions, so doubling the total work scales each dimension by $\sqrt[3]{2}$ (≈ 1.26), and the perfect-cube domain count restricts MPI (one rank per GPU) to perfect-cube GPU counts. Charm++ can decouple the chare count from the GPU count, so we report Charm++ results across the full scaling range.

a) Weak scaling on NVIDIA: We evaluate weak scaling from 1 to 32 GPUs (8 nodes), starting from a $256 \times 256 \times 256$ grid and growing the grid dimensions as described earlier to keep the per-GPU workload constant. The integral ODF configurations closely track MPI across the full range, mirroring the trends observed for Jacobi2D and MiniMD. The fractional ODF configurations, in contrast, are consistently slower at most scales, suggesting that imbalance introduced by chare placement and mapping is at fault.

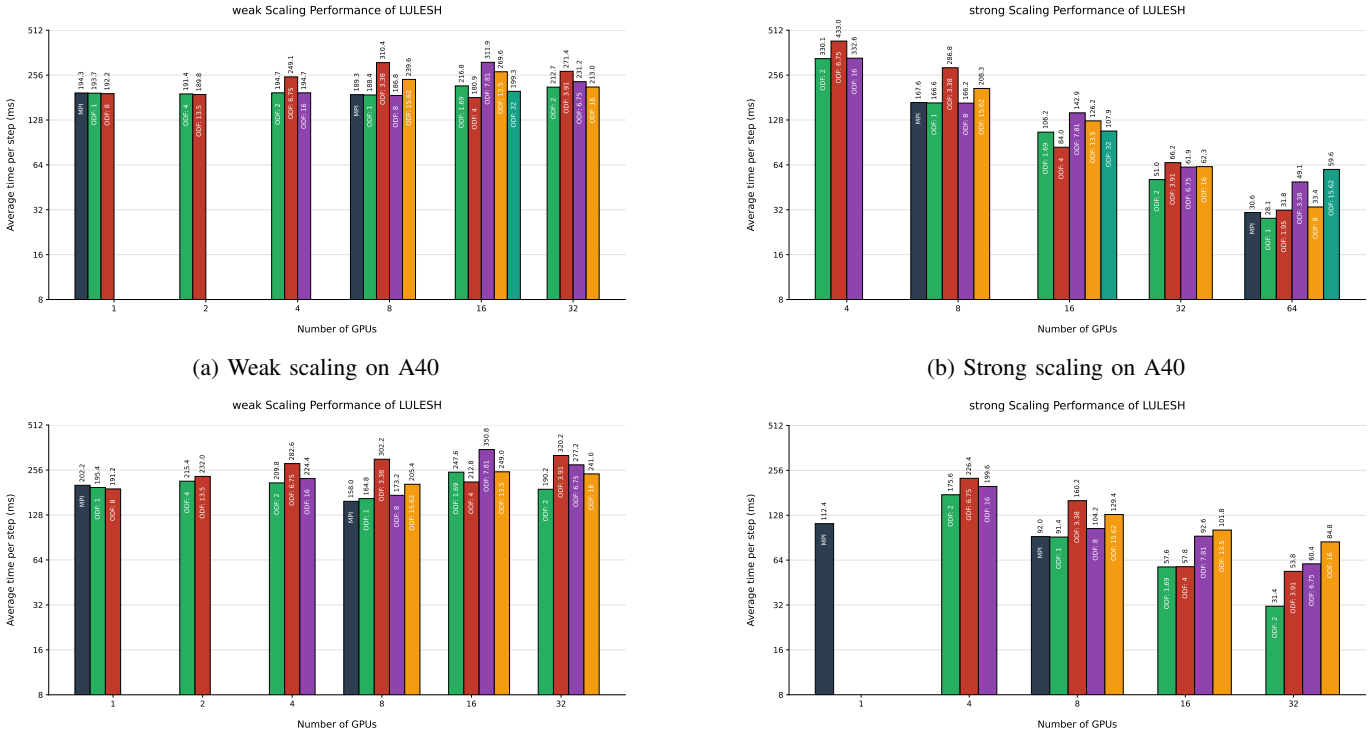
b) Strong scaling on NVIDIA: For strong scaling, we fix the domain at $480 \times 480 \times 480$ and scale from 4 GPUs (1 node) to 64 GPUs (16 nodes). At smaller scales, the Charm++ configurations again track MPI closely across ODFs. At 32 and 64 GPUs, however, the higher integral ODFs degrade slightly relative to intermediate ODFs, as in MiniMD: the shrinking per-GPU problem size leaves less computation to amortize the overheads of high overdecomposition. Fractional ODF configurations remain consistently slower throughout.

c) Weak scaling on AMD: On AMD, we weak-scale from 1 to 32 GPUs (4 nodes) with a base domain of $320 \times 320 \times 320$. Figure 13c shows Charm++ performing comparably to MPI, with some loss at higher ODFs. We attribute this in part to the event-based IPC signaling mechanism used for intra-node HIP communication, which currently requires additional synchronization.

d) Strong scaling on AMD: Strong scaling on AMD uses the same fixed $480 \times 480 \times 480$ domain as the NVIDIA runs, scaling from 4 to 32 GPUs (4 nodes). The behavior mirrors the AMD weak-scaling results, with Charm++ incurring noticeable overhead relative to MPI at larger ODFs.

VI. Conclusion and Future Work

This paper examined whether overdecomposition, a technique whose benefits are well established on CPU-based systems, can be supported efficiently and portably on modern



(a) Weak scaling on A40

(b) Strong scaling on A40

(c) Weak scaling on MI250X

(d) Strong scaling on MI250X

Fig. 13: LULESH weak and strong scaling on A40 and MI250X GPUs.

GPU platforms. We extended Charm++ with a portable GPU runtime that pairs Kokkos for execution portability with an LCI-based communication layer operating over both ibverbs and libfabric, so that a single-source overdecomposed application runs unchanged on NVIDIA and AMD systems.

Microbenchmark analysis showed that the principal cost of fine-grained GPU execution resides on the host rather than the device: kernel launch and completion detection serialize within a process context, dominated by the submission path on the GH200 and the event-query path on the MI250X. These observations motivated a hierarchical execution model with multiple processes per GPU and multiple worker threads per process, parallelizing kernel submission and completion detection, complemented by a flag-based mechanism that eliminates serialized event queries. The communication layer routes each message through the transport suited to its endpoints, using device-to-device copies within a process, staged IPC within a node, and one-sided RDMA across nodes, without staging data through the host.

Task Bench isolates the computational side of these mechanisms in an end-to-end task graph: overdecomposition with asynchronous completion detection reduces the minimum effective task granularity from 26–37μs to 10.5μs on a single PE per GPU, and parallel submission across multiple PEs reduces it further to 4μs. The three mini-applications confirm that these improvements carry over to application settings, with overdecomposed Charm++ configurations performing on par with their MPI counterparts across weak and strong scaling on both A40 and MI250X systems.

More fundamentally, this work establishes the potential of overdecomposition on GPUs. Although the advantages of overdecomposition have been independently demonstrated, especially on CPU-based systems, realizing and demonstrating its utility on GPU nodes, in particular for dynamic load balancing and malleable execution, will require efficient chare migration and accurate load estimation on accelerators, each with its own challenges. The runtime presented here provides the foundation upon which these capabilities can be built. We also plan to extend the runtime to Intel GPUs: Kokkos's SYCL backend covers kernel execution, and our effort will focus on a Level Zero IPC transport path, with inter-node GPU communication to follow as LCI adds RDMA support for Intel devices.

VII. Acknowledgments

This work used Delta at NCSA through allocation ASC-050025 from the ACCESS program, supported by NSF grants #2138259, #2138286, #2138307, #2137603, and #2138296. Delta is supported by NSF award OAC 2005572 and the State of Illinois. This research also used resources of the Oak Ridge Leadership Computing Facility, a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725, and Vista at the Texas Advanced Computing Center (TACC) at The University of Texas at Austin under project MCB24004. This work is supported in part by the IBM-Illinois Discovery Accelerator Institute (IIDAI).

References

[1] Bilge Acun, Abhishek Gupta, Nikhil Jain, Akhil Langer, Harshitha Menon, Eric Mikida, Xiang Ni, Michael Robson, Yanhua Sun, Ehsan Totoni, Lukasz Wesolowski, and Laxmikant Kale. Parallel Programming with Migratable Objects: Charm++ in Practice. SC, 2014.

[2] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. In Henk Sips, Dick Epema, and Hai-Xiang Lin, editors, *Euro-Par 2009 Parallel Processing*, pages 863–874, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: expressing locality and independence with logical regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, Washington, DC, USA, 2012. IEEE Computer Society Press.

[4] David A. Beckingsale, Jason Burmark, Rich Hornung, Holger Jones, William Killian, Adam J. Kunen, Olga Pearce, Peter Robinson, Brian S. Ryujin, and Thomas RW Scogland. Raja: Portable performance for large-scale scientific applications. In *2019 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 71–81, 2019.

[5] Aditya Bhosale, Kavitha Chandrasekar, Laxmikant Kale, and Sara Kokkila-Schumacher. An elastic job scheduler for hpc applications on the cloud. In *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC Workshops '25, page 151–162, New York, NY, USA, 2025. Association for Computing Machinery.

[6] Aditya Bhosale, Advait Tahilyani, Laxmikant Kale, and Sara Kokkila-Schumacher. Towards an adaptive runtime system for cloud-native hpc, 2026.

[7] Aurelien Bouteiller, Thomas Herault, Qinglei Cao, Joseph Schuchart, and George Bosilca. PaRSEC: Scalability, flexibility, and hybrid architecture support for task-based applications in ECP. *The International Journal of High Performance Computing Applications*, 2025.

[8] Jaemin Choi, David F. Richards, and Laxmikant V. Kale. Improving scalability with gpu-aware asynchronous tasks. In *2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 569–578, 2022.

[9] Gregor Dais, Mikael Simberg, Auriane Reverdell, John Biddiscombe, Theresa Pollinger, Hartmut Kaiser, and Dirk Pfluger. Beyond Fork-Join: Integration of Performance Portable Kokkos Kernels with HPX . In *2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 377–386, Los Alamitos, CA, USA, June 2021. IEEE Computer Society.

[10] H. Carter Edwards, Christian R. Trott, and Daniel Sunderland. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[11] Abhishek Gupta, Bilge Acun, Osman Sarood, and Laxmikant V. Kale. Towards Realizing the Potential of Malleable Parallel Jobs. In *Proceedings of the IEEE International Conference on High Performance Computing*, HiPC '14, Goa, India, December 2014.

[12] Michael A. Heroux, Douglas W. Doerfler, Paul S. Crozier, James M. Willenbring, H. Carter Edwards, Alan Williams, Mahesh Rajan, Eric R. Keiter, Heidi K. Thornquist, and Robert W. Numrich. Improving performance via mini-applications. Technical Report SAND2009-5574, Sandia National Laboratories, 2009.

[13] John K. Holmen, Marta García, Allen Sanderson, Abhishek Bagusetty, and Martin Berzins. Lessons learned and scalability achieved when porting uintah to doe exascale systems. In *Euro-Par 2024: Parallel Processing Workshops: Euro-Par 2024 International Workshops, Madrid, Spain, August 26–30, 2024, Proceedings, Part I*, page 231–242, Berlin, Heidelberg, 2024. Springer-Verlag.

[14] Nikhil Jain, Eric Bohm, Eric Mikida, Subhasish Mandal, Minjung Kim, Prateek Jindal, Qi Li, Sohrab Ismail-Beigi, Glenn Martyna, and Laxmikant Kale. Openatom: Scalable ab-initio molecular dynamics with diverse capabilities. In *International Supercomputing Conference*, ISC HPC '16, 2016.

[15] Pritish Jetley, Filippo Gioachin, Celso Mendes, Laxmikant V. Kale, and Thomas R. Quinn. Massively parallel cosmological simulations with ChaNGa. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008.

[16] Hartmut Kaiser, Patrick Diehl, Adrian S. Lemoine, Bryce Adelstein Lelbach, Parsa Amini, Agustín Berge, John Biddiscombe, Steven R. Brandt, Nikunj Gupta, Thomas Heller, Kevin Huck, Zahra Khatami, Alireza Kheirkhahan, Auriane Reverdell, Shahrzad Shirzad, Mikael Simberg, Bibek Wagle, Weile Wei, and Tianyi Zhang. Hpx - the c++ standard library for parallelism and concurrency. *Journal of Open Source Software*, 5(53):2352, 2020.

[17] L.V. Kalé and S. Krishnan. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In A. Paepcke, editor, *Proceedings of OOPSLA'93*, pages 91–108. ACM Press, September 1993.

[18] Ian Karlin, Jeff Keasler, and Rob Neely. LULESH 2.0 updates and changes. Technical Report LLNL-TR-641973, Lawrence Livermore National Laboratory, Livermore, CA, August 2013.

[19] Khronos SYCL Working Group. SYCL 2020 specification. https://registry.khronos.org/SYCL/specs/sycl-2020/html/sycl-2020.html, 2023. Khronos Group.

[20] Lawrence E. Kidder, Scott E. Field, Francois Foucart, Erik Schnetter, Saul A. Teukolsky, Andy Bohn, Nils Deppe, Peter Diener, François Hébert, Jonas Lippuner, Jonah Miller, Christian D. Ott, Mark A. Scheel, and Trevor Vincent. SpECTRE: A task-based discontinuous Galerkin code for relativistic astrophysics. *Journal of Computational Physics*, 335:84–114, 2017.

[21] E. Meneses, Xiang Ni, Gengbin Zheng, C.L. Mendes, and L.V. Kale. Using migratable objects to enhance fault tolerance schemes in supercomputers. *Parallel and Distributed Systems, IEEE Transactions on*, 26(7):2061–2074, July 2015.

[22] James C. Phillips, Rosemary Braun, Wei Wang, James Gumbart, Emad Tajkhorshid, Elizabeth Villa, Christophe Chipot, Robert D. Skeel, Laxmikant Kalé, and Klaus Schulten. Scalable molecular dynamics with NAMD. *Journal of Computational Chemistry*, 26(16):1781–1802, 2005.

[23] Elliott Slaughter, Wei Wu, Yuankun Fu, Legend Brandenburg, Nicolai Garcia, Wilhem Kautz, Emily Marx, Kaleb S. Morris, Qinglei Cao, George Bosilca, Seema Mirchandaney, Wonchan Lee, Sean Treichler, Patrick McCormick, and Alex Aiken. Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. IEEE Press, 2020.

[24] Rohan Yadav, Joseph Guman, Sean Treichler, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. On the duality of task and actor programming models, 2025.

[25] Jiakun Yan and Marc Snir. Lci: a lightweight communication interface for efficient asynchronous multithreaded communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '25, page 1043–1059, New York, NY, USA, 2025. Association for Computing Machinery.

APPENDIX
ARTIFACT DESCRIPTION

PART 1: OVERVIEW OF CONTRIBUTIONS AND ARTIFACTS

A. Paper's Main Contributions

- C_1 A portable GPU runtime for Charm++ built on LCI and Kokkos, supporting NVIDIA and AMD GPUs from a single source.
- C_2 A hierarchical execution model with multiple processes per GPU and multiple worker threads per process, complemented by a flag-based completion detection mechanism.
- C_3 An empirical study of overdecomposition cost on NVIDIA and AMD GPUs.

B. Computational Artifacts

Artifact	Contributions	Paper elements
A_1: Microbenchmark suite	C_2, C_3	Figures 1–5, 7, 8; Table I
A_2: Task Bench	C_1, C_2	Figure 10
A_3: Mini-applications	C_1, C_2	Figures 11–13

All artifacts run on the extended Charm++ runtime with the reconverse layer (branch `paw-atm26` of both). Installation instructions and the exact commands for every experiment are in the repository READMEs:

- https://github.com/charmplusplus/charm/tree/paw-atm26
- https://github.com/charmplusplus/reconverse/tree/paw-atm26
- https://github.com/charmplusplus/reconverse/wiki/How-to-run-Charm-with-reconverse
- https://github.com/charmplusplus/task-bench
- https://github.com/charmplusplus/jacobi2d_kokkos
- https://github.com/charmplusplus/miniMD
- https://github.com/charmplusplus/kokkos-miniapps

PART 2: ARTIFACT IDENTIFICATION

Artifact A_1: Microbenchmark suite

A. Relation to Contributions: The microbenchmarks behind the empirical study (C_3): kernel decomposition (Figures 1, 8), device communication paths (Figure 2), launch overhead and rate for threads and processes (Figures 3, 4), event query cost (Table I), flag versus event completion detection (Figure 5), and the process/thread split (Figure 7). They motivate and evaluate the mechanisms of C_2.

B. Expected Results: Launch overhead on GH200 grows with thread count while the MPS aggregate stays flat near 0.95M launches/s; MI250X event queries cost about 10 μs and serialize (Table I); flag detection overtakes event polling beyond 8 threads; the split benchmark reproduces the regime structure of Figure 7. These substantiate the claim that fine-grained overheads are host-side, vendor-specific, and addressable by the hierarchical model.

C. Expected Reproduction Time (in minutes): Setup 60 per system; execution 90 (NVIDIA), 45 (AMD), plus 120 for the split benchmark; analysis 10.

D. Artifact Setup:

Hardware: one exclusive GH200 node of TACC Vista and one exclusive MI250X node of OLCF Frontier.

Software: the Charm++ runtime above; CUDA 12.5 (Vista) or ROCm 6.4.2; Python 3 with matplotlib; NVIDIA MPS.

Datasets/Input: none; the benchmarks are synthetic.

Installation & Deployment: see the repository READMEs.

E. Artifact Evaluation: T_1 (build) $\rightarrow$ T_2 (sweep scripts, one per figure, writing RESULT lines to logs) $\rightarrow$ T_3 (plot scripts). Thread/process counts sweep powers of two; kernel grains sweep 1 to 100 μs and message sizes 8 B to 16 MB, spanning the overhead- to bandwidth-dominated regimes; the split benchmark uses three repetitions per cell (run-to-run variance 5–8%).

F. Artifact Analysis: Python scripts parse the logs, aggregate repetitions by median, and emit the figures.

Artifact A_2: Task Bench

A. Relation to Contributions: Measures the end-to-end effect of the execution model (C_2) in the portable runtime (C_1) via METG, against two MPI references that differ in completion handling.

B. Expected Results: METG of 26–37 μs without overdecomposition; 10.5 μs with 8 chares per GPU on one PE; near 4 μs with 4 or more PEs; MPI references near 11.5 and 5.6 μs (Figure 10, within about 10%).

C. Expected Reproduction Time (in minutes): Setup 45; execution 60 per configuration (three configurations); analysis 10.

D. Artifact Setup:

Hardware: up to four A40 nodes of NCSA Delta.

Software: as A_1 with CUDA 12.8 (Delta); Task Bench is in the `task-bench` repository.

Datasets/Input: none; task graphs are generated.

Installation & Deployment: see the repository README.

E. Artifact Evaluation: T_1 (build) $\rightarrow$ T_2 (METG sweeps at graph widths 4, 32, 128, halving per-task work until efficiency collapses, five repetitions per point; Charm++ additionally sweeps PEs per GPU) $\rightarrow$ T_3 (METG computation).

F. Artifact Analysis: A Python script computes efficiency from the logs, interpolates the 50% crossing to obtain METG, and plots Figure 10.

Artifact A_3: Mini-applications

A. Relation to Contributions: Evaluates the runtime (C_1, C_2) at application scale: Jacobi2D, MiniMD, and LULESH in Charm++/Kokkos against MPI/Kokkos references.

B. Expected Results: Integral ODF configurations track the MPI references within a few percent in weak and strong scaling on both systems, with the deviations discussed in the paper, substantiating that overdecomposition adds little overhead at application granularity.

C. Expected Reproduction Time (in minutes): Setup 60 per system; execution 240 per system; analysis 10.

D. Artifact Setup:

Hardware: up to 16 A40 nodes of NCSA Delta (64 GPUs) and 4 MI250X nodes of OLCF Frontier (32 GCDs).

Software: as A_1/A_2, plus Kokkos 4.7.01; application repositories listed in Part 1.

Datasets/Input: none; problem sizes are command-line parameters.

Installation & Deployment: see the repository READMEs.

E. Artifact Evaluation: T_1 (build) $\rightarrow$ T_2 (weak and strong scaling sweeps over ODF $\in \{1, 2, 4, 8, 16\}$, 100 iterations per run, problem sizes as in the paper) $\rightarrow$ T_3 (plots).

F. Artifact Analysis: A Python script collects per-step times and renders the bar charts of Figures 11–13.