frontiers | Frontiers in High Performance Computing

TYPE Original Research
PUBLISHED 19 July 2024
DOI 10.3389/fhpcp.2024.1417040

OPEN ACCESS

EDITED BY
Geoffrey Fox,
University of Virginia, United States

REVIEWED BY
Francisco F. Rivera,
University of Santiago de Compostela, Spain
Eero Vainikko,
University of Tartu, Estonia

*CORRESPONDENCE
Polykarpos Thomadakis
✉ pthom001@odu.edu

RECEIVED 13 April 2024
ACCEPTED 02 July 2024
PUBLISHED 19 July 2024

CITATION
Thomadakis P and Chrisochoides N (2024) Runtime support for CPU-GPU high-performance computing on distributed memory platforms.
Front. High Perform. Comput. 2:1417040.
doi: 10.3389/fhpcp.2024.1417040

Runtime support for CPU-GPU high-performance computing on distributed memory platforms

Polykarpos Thomadakis* and Nikos Chrisochoides

Center for Real-Time Computing, Old Dominion University, Norfolk, VA, United States

Introduction: Hardware heterogeneity is here to stay for high-performance computing. Large-scale systems are currently equipped with multiple GPU accelerators per compute node and are expected to incorporate more specialized hardware. This shift in the computing ecosystem offers many opportunities for performance improvement; however, it also increases the complexity of programming for such architectures.

Methods: This work introduces a runtime framework that enables effortless programming for heterogeneous systems while efficiently utilizing hardware resources. The framework is integrated within a distributed and scalable runtime system to facilitate performance portability across heterogeneous nodes. Along with the design, this paper describes the implementation and optimizations performed, achieving up to 300% improvement on a single device and linear scalability on a node equipped with four GPUs.

Results: The framework in a distributed memory environment offers portable abstractions that enable efficient inter-node communication among devices with varying capabilities. It delivers superior performance compared to MPI+CUDA by up to 20% for large messages while keeping the overheads for small messages within 10%. Furthermore, the results of our performance evaluation in a distributed Jacobi proxy application demonstrate that our software imposes minimal overhead and achieves a performance improvement of up to 40%.

Discussion: This is accomplished by the optimizations at the library level and by creating opportunities to leverage application-specific optimizations like over-decomposition.

KEYWORDS

parallel computing, distributed computing, GPGPU programming, runtime systems, heterogeneous systems, high-performance computing

1 Introduction

The recent slowdown in Moore's Law is leading to large-scale disruptions in the computing ecosystem. Users and vendors are transitioning from utilizing computing nodes of relatively homogeneous CPU architectures to systems led by multiple GPU devices per node. This trend is expected to continue in the foreseeable future, incorporating many more types of heterogeneous devices, including FPGAs, System-on-Chips (SoCs), and specialized hardware for artificial intelligence (Ang et al., 2021). The new computing ecosystem sets the basis to significantly improve performance, energy efficiency, reliability, and security; thus, high-performance computing (HPC) systems are adapted and optimized for traditional and modern workloads.

Exploiting extreme heterogeneity requires new techniques and abstractions that handle the increasing complexity in productivity, portability, and performance. The new methods should allow users to express their applications' workflow uniformly, hiding the peculiarities of the underlying architecture while handling concerns arising from performance portability. One such concern is managing data on various devices. In most cases, data need to be transferred among devices to execute kernels optimized explicitly for an accelerator; thus, a framework needs to allocate the respective memory, find the devices involved in such a transaction, initiate the transfer, and monitor its progress. The coherence of the same data in different devices is also an issue. One needs to guarantee that the application will always use the most recent version of data, no matter which device. Moreover, since these operations are substantial overheads, they should happen asynchronously and overlap with valuable work, further increasing complexity.

Another concern is the orchestration of task (computation) execution. Tasks should only start asynchronous executions after the respective data have been moved to the target device and only when they do not conflict with other tasks, requiring lightweight synchronization. Task scheduling and load balancing should also be a significant concern to keep the available devices saturated and fully utilize them. The schedulers should be aware of each device's load and the data locality of each task to designate where each computation should occur efficiently. Finally, a framework that handles all these concerns should provide a friendly, high-level interface for applications but also expose low-level access that allows experts to optimize their applications for their specific needs. Moreover, having the option of lower-level access is crucial for distributed memory frameworks to use them on multiple nodes efficiently.

Utilizing and orchestrating data movement and task execution on multiple heterogeneous nodes increases the number of issues that need to be tackled. Thus, a complete runtime framework should also facilitate the seamless use of distributed heterogeneous nodes using the same approach of abstractions for data and workload independent of the underlying hardware and have tight integration with the performance portability layer used in a single node. Current trends in HPC follow the programming model of MPI+CUDA, which leads to complicated code, suboptimal performance, or both. Users that follow this approach need to explicitly transfer data between the host and the device before sending/receiving to/from a remote node. Moreover, they will need to use asynchronous operations for both memory transfer operations (host-GPU and network) and overlap them to avoid wasting cycles. This leads to the concern of correctness and synchronization involving the two types of data transfers and asynchronous kernel invocations. And even if one manages to handle all those correctly, they would have an application that only operates efficiently (or at all) for the specific hardware it was developed. Thus, a distributed framework that natively incorporates and abstracts heterogeneous nodes is the only way to create applications that scale independently of the hardware in which they were implemented.

In Thomadakis et al. (2022), we presented the most recent evolution of Parallel Runtime Environment for Multicomputer Applications (PREMA), a scalable runtime system for distributed homogeneous platforms. It uses high-level abstractions to simplify distributed programming for dynamic and irregular applications (Garner et al., 2024). In this work, we extend PREMA to support seamless, efficient, and performance-portable development of distributed applications on heterogeneous nodes. First, we introduce a heterogeneous tasking framework to optimize the parallel execution of heterogeneous tasks on a single node. The tasking framework provides a programming model that automatically leverages heterogeneous devices. In contrast to other systems, our framework does not require the application to choose a device where a task should run; instead, the application only picks a device type, and the framework is responsible for scheduling the task to the optimal computing device. Next, we integrate PREMA with the heterogeneous tasking framework and enable it to manage and utilize heterogeneous nodes uniformly. Along with the design and implementation of the final product, we present optimizations that contribute to achieving high performance. The evaluation results with microbenchmarks and a proxy application show that our system incurs low overhead with scalable performance.

1.1 Parallel runtime environment for multicore applications

PREMA (Chrisochoides, 1995, 1998) is a software system designed to provide runtime support for large-scale, dynamic, irregular and data-intensive applications like parallel n-body computations (Balasubramaniam et al., 2004) and parallel adaptive unstructured mesh generation (Nave et al., 2004; Chrisochoides, 2005, 2016; Chernikov and Chrisochoides, 2008; Foteinos and Chrisochoides, 2014; Drakopoulos et al., 2019; Tsolakis et al., 2022) as opposed to earlier runtime systems (Fox et al., 1993) designed for more regular compute-intensive HPC codes (Bozkus et al., 1993; Chrisochoides et al., 1994; Parashar et al., 1994; Baden et al., 1999). It utilizes a 2-level parallelism approach that employs Message Passing Interface (MPI) for inter-node communication and Pthreads or Argobots (Seo et al., 2016) for intra-node coordination. The system uses the construct of mobile objects, which are globally addressable, location-independent containers (Chrisochoides et al., 2000; Fedorov and Chrisochoides, 2004) that hold application data (Chrisochoides and Hawblitzel, 1998). Mobile objects enable the mobile object-driven (MOD) programming model (Barker et al., 2004), which facilitates interactions between local or remote mobile objects through remote method invocations (called handlers) (von Eicken et al., 1992; Chrisochoides et al., 1997; Barker et al., 2002). Handlers can be invoked on mobile objects uniformly, regardless of whether their data are local or remote, effectively providing the user with a virtual global namespace throughout the distributed system by automatically generating a local task (if on the same node) or issuing a message (if on a remote node) depending on the location of the handler target. This approach abstracts the complexity of work scheduling, load balancing, communication overhead, and out-of-core computing, allowing applications to utilize available computing power without explicit concurrency handling (Chrisochoides, 1996; Kot et al., 2011;

Thomadakis et al., 2018, 2022; Garner et al., 2019; Thomadakis and Chrisochoides, 2023). Meanwhile, by requiring explicitly issuing handler invocation requests to operate on potentially remote data, PREMA makes it easier for the users to understand and reason about the underlying costs in order to design their algorithms efficiently. Figure 1 shows an example of the MOD model.

PREMA extracts shared-memory parallelism by running non-conflicting handlers concurrently while implicitly migrating mobile objects among computing nodes to provide distributed-memory load balancing. Thus, the hardware memory spaces and processing elements are virtualized, allowing inter-handler parallelism (multiple handlers running in parallel) and sharing mobile object workload for threads in the same node. Additionally, it offers a module for easy experimentation and development of new 2-level load balancing/scheduling policies that handle both shared and distributed memory data and work distribution.

1.2 Contributions

This paper presents an effort to address the challenges of efficiently utilizing distributed heterogeneous computing nodes in a portable and performant way. It introduces a heterogeneous tasking framework as an extensible layer of portability. In addition, it presents the integration of the framework within a distributed memory system to produce a design that natively handles distributed nodes equipped with heterogeneous devices. The major contributions of this paper are as follows.

- A new design and implementation of a heterogeneous tasking framework for the development of performance portable applications that can scale from single-core to multi-core, multi-device (*CPUs, GPUs*) platforms efficiently, *without any code refactoring*.
- A novel integration of a distributed runtime with the heterogeneous tasking framework to provide an end-to-end solution that scales over distributed heterogeneous computing nodes while exposing a high-level and abstract programming model.
- A series of memory, scheduling, and threading performance optimizations that achieve significant improvements, up to 300% on a single GPU and linear scalability on a multi-GPU platform, that are directly applicable to similar systems and applications.
- Demonstration of up to 40% speedup on an end-to-end distributed, heterogeneous proxy application (e.g., Jacobi solver) by utilizing the new runtime framework in combination with widely studied optimizations like over-decomposition (Chrisochoides, 1996).

2 Related work

Several systems have been adapted to efficiently utilize GPUs in their workflow, while new ones have emerged trying to create new standards for their use. X10 (Charles et al., 2005) and its successors Habanero Java and C (Cavé et al., 2011; Majeti and Sarkar, 2015) are parallel programming languages based on Java and C++ that expose a PGAS memory model. Instead of an explicit SPMD model, programmers are given a single entry point from which they can explicitly allocate data and tasks on local and remote nodes. The compiler then takes care of spawning multiple processes and distributing data efficiently. They introduce the abstraction of places that are independent and disjoint pieces of virtual addresses that can map to different pieces of hardware. In-place parallelism is achieved through issuing tasks, while remote computations on other places require explicitly targeting a place. Support for GPUs is provided as another implementation of places, but data allocations and transfers have to be handled explicitly; no implicit construct is provided. Chapel (Chamberlain et al., 2007) is in many aspects similar to X10. It provides its own flavor for abstract virtual addresses called locales and, as X10, starts from a single entry point from where tasks and data are distributed. GPUs are handled uniformly, i.e., data transfers are automatically generated and monitored; however, the user has to assign work to each GPU explicitly. Like X10, data mappings are done statically from the compiler and cannot change, which raises the same issues of load balancing and bad fit for irregular applications. Charm++ (Kale and Krishnan, 1993) is a runtime similar to PREMA, sharing some of its abstractions for mobile objects which are called chares in that context. It is based on C++ but requires a dedicated compiler that can generate the object marshaling methods, as well as generate the code necessary to invoke methods remotely. Its support for GPUs, however, is limited, mainly providing interfaces to allow time-slicing between GPU operations of different chares. Submitting tasks to GPUs needs to be explicitly handled by the user, who should allocate memory, issue data transfers, and handle scheduling in case of multiple GPUs. HPX (Kaiser et al., 2014) also lets the users explicitly handle issues like requesting memory transfers, managing device platforms, task allocations, and work queues to optimize performance. In contrast, PREMA provides a uniform abstraction for heterogeneous tasks and data and implicitly handles scheduling, load balancing, and latency overlapping independently of the target device backend. StarPU (Augonnet et al., 2011), OmpSs (Duran et al., 2011), and ParSec (Bosilca et al., 2012) offer different high-level approaches for efficiently utilizing distributed heterogeneous systems. However, their programming model is better suited for applications whose workflow follows a regular pattern that can be inferred mostly statically. On the other hand, PREMA adopts a dynamic, message-driven programming model that is more suitable for irregular applications. Legion (Bauer et al., 2012) is a high-level task-based heterogeneous runtime system for distributed memory architectures that separates the application workflow from mapping computations and data to hardware. It utilizes the primitive concepts of logical regions as the piece of data to present data organization and expose task-data interactions. Even though it is a powerful system, it requires a lot of effort and code rewriting to port existing applications on top of it and its design better conforms to structured data.

On the shared memory space, multiple efforts have emerged and are included for completeness even though they do not handle distributed memory systems like PREMA. SYCL and DPC++/oneAPI (Ashbaugh et al., 2020), as well as the newest version of OpenMP, are recent attempts to provide performance portable interfaces in modern C++ that can target heterogeneous devices. However, users still need to handle load balancing,

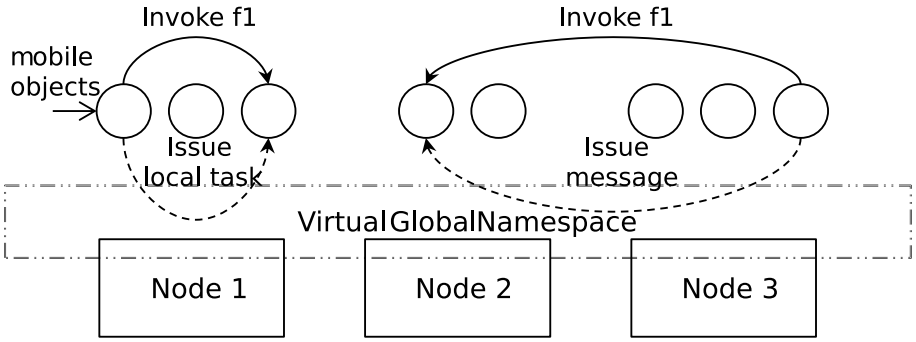

FIGURE 1
PREMA's mobile object driven (MOD) model. Applications are expressed as method invocations between local or remote mobile objects. A virtual global namespace provides a uniform high-level interface to issue local or remote work. The runtime system is responsible for running a task locally or sending an active message depending on the location of the target mobile object. Figure adapted from Thomadakis and Chrisochoides (2023).

scheduling, and work queues for multi-device systems and need to combine them with another runtime solution that targets distributed nodes. RAJA (Beckingsale et al., 2019) provides an interface of C++ templates that capture platform-specific constructs and allow users to write kernels with a single source that can run on different devices. RAJA can then map those kernels to the specific platform implementation requirements and handle data transfers. Even though it abstracts the application code to a high degree, users need to explicitly issue tasks to distinct accelerators to utilize multiple devices and explicitly handle task dependencies. Kokkos (Carter Edwards et al., 2014) provides similar capabilities as RAJA, including abstractions for data objects, layouts, and portable kernels using a single source code. In addition, Kokkos provides the ability for task-based parallelism allowing for the creation of task dependencies through the construction of DAGs. Moreover, it has more default implementations for decisions that depend on the application, e.g., automatic data layout formatting, kernel block size inference, etc. Like RAJA, Kokkos does not explicitly handle multi-GPU scheduling; the application must submit tasks to different GPUs to saturate multi-GPU platforms. TaskflFlow (Huang et al., 2022) is another shared memory heterogeneous runtime that uses an expressive task graph programming model to assist developers in the implementation of parallel and heterogeneous decomposition strategies on a heterogeneous computing platform. It provides a plethora of task-graph flow-handling interfaces but it expects the users to explicitly provide the task graph, allocate device memory, handle data copies, and even assign tasks to different execution streams to promote device parallelism, using vendor-specific APIs. In our work, all of these concerns are handled automatically by the runtime framework in an abstract and vendor-independent way.

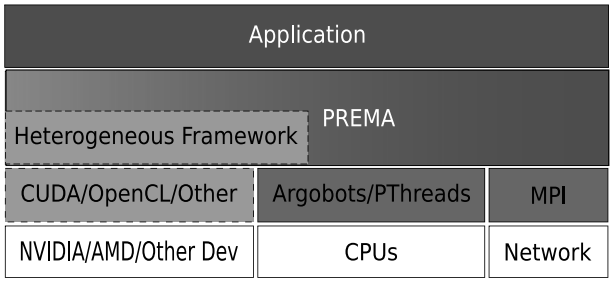

FIGURE 2
A high-level representation of the heterogeneity-aware PREMA. The hardware devices/interfaces stand on the lower level and are utilized by integrating PREMA with MPI, PThreads, and Argobots (CPU-only; see Thomadakis et al., 2022 and Thomadakis and Chrisochoides, 2023), and the heterogeneous tasking framework (in the current work). On top of that stands the application, which leverages these capabilities through a simple but powerful interface.

3 Design and implementation

Following the principle of separation of concerns, a new abstract compatibility layer is introduced, allowing PREMA to access different heterogeneous devices uniformly. This layer is implemented as a stand-alone heterogeneous tasking framework that handles all concerns arising from the co-existence of multiple types of devices. PREMA integrates this framework as the preferred way to interact with heterogeneous devices. Thus, it can be easily extended to utilize more device types without needing to modify its implementation, apart from this low-level compatibility layer. Moreover, PREMA exposes some of these capabilities wrapped in a high-level interface, allowing users to utilize such devices in a controlled and safe way. Figure 2 shows a high-level representation of the software stack and how the different layers interact. In the following sections, first, we present the heterogeneous framework layer and its capabilities in detail; then, we focus on its integration with PREMA to provide a distributed, heterogeneity-aware runtime.

3.1 Heterogeneous tasking framework

The programming model of the heterogeneous tasking framework builds upon two simple abstractions: the heterogeneous objects (*hetero_objects*) and heterogeneous tasks (*hetero_tasks*). A *hetero_object* uniformly represents a user-defined data object residing on one or more computing devices of a heterogeneous

compute node (e.g., CPUs, GPUs). Applications treat such objects as opaque containers for data without being aware of their physical location. A *hetero_task* encapsulates a non-preemptive computing kernel that runs to completion and implements a medium-grained parallel computation. Like hetero_objects, hetero_tasks are defined and handled by the application uniformly, independent of the device they will execute on.

Figure 3 shows an example of a DGEMM implementation using the tasking framework. The presented example shows a DGEMM execution request for the GPU; however, by only changing the device target in line 53, one can target a different device without touching the rest of the code. In lines 28–30, the application requests from the runtime to reserve memory for the data involved in the computation along with the types and counts needed using the *hetero_object* abstraction (described in detail later). At this point, the actual buffers might be allocated in the CPU, GPU, or not allocated at all. Applications can request read and/or write access to the host (CPU) side using the asynchronous method *request(bool,bool)*(lines 33–34), which once complete, guarantees that the underlying buffers are available and consistent. Once access to the underlying buffers is no longer needed, a call to method *release()* returns control of the corresponding data back to the runtime. Note that the runtime guarantees that no other device/thread can mutate these data until they are released from the application side. Another, more efficient, way to manipulate hetero_objects is through *hetero_tasks*(lines 45–56) as described in Section 3.1.2.

The kernel() macro on the top of the listing will expand to CUDA, OpenCL, and/or other defined backends to fit the vendor-provided implementation. In this specific example, *device_global* qualifier is used to satisfy OpenCL's requirement to denote device data as residing in global memory as opposed to local or constant and is ignored when compiling for CUDA or CPUs. Similarly, *kernel_group_id*, *kernel_local_id* keywords map to the respective *blockId*/*get_group_id()*, *threadId*/*get_local_id()* identities of CUDA, OpenCL and a task ID, loop index in CPUs.

3.1.1 Heterogeneous objects

Handling copies of the same data on different heterogeneous devices can lead to error-prone and difficult-to-maintain application code. In general, applications need to manage data transfers among them, use the correct pointer for the respective device, and keep track of their coherence. A hetero_object is an abstraction that automatically handles such concerns, maintaining the different copies of the same data in a single reference. The underlying system controls hetero_objects to guarantee that the most recent version of the data will be available at the target device when needed. For example, accessing an object currently resident on the CPU from a GPU would automatically trigger the transfer of the underlying data from the host to the respective device. In the same manner, accessing the same object from a different device would initiate a transfer from the GPU to that device, potentially after first staging the data at the host. Finally, the runtime system guarantees data coherence among computing devices, keeping track of up-to-date or stale copies and handling them appropriately.

The memory captured by a hetero_object should mainly be accessed and modified through hetero_tasks for optimal performance. However, the application can also explicitly request access to the underlying data on the host after specifying the type of access requested to maintain coherence. This method will trigger (if needed) an asynchronous transfer from the device with the most recent version of the data and immediately return a future. The future is a construct that acts as a placeholder for the data that will be available once the operation is completed in the future. It allows for querying the transfer status, synchronizing and providing access to the raw data once the transfer has been completed. In this state, the data of the hetero_object are guaranteed to remain valid on the host side, preventing tasks that would alter them from executing until the user explicitly releases their control back to the runtime system. Since the application has no direct access to the memory allocated to different devices, our framework monitors the memory usage of each device. When a device's memory is close to being depleted, the runtime system will automatically start offloading some of the user's data to the host or other devices. We currently use a Least Recently Used (LRU) policy to determine which hetero_object should be offloaded to free a device's memory. LRU is preferred due to its simple implementation and maintenance and is usually quite effective. However, an exploration of eviction policies is out of the scope of this work. An application can explicitly request to remove a hetero_object from all devices to help the runtime clean up some memory; otherwise, a hetero_object will be freed when going out of scope. In both cases, the hetero_object will only be removed once no tasks and other operations are referencing it.

3.1.2 Heterogeneous tasks

Heterogeneous tasks (hetero_tasks) are opaque structures that consolidate the parameters characterizing a computational task. Through a hetero_task, applications define the kernel to execute (Figure 3 line 56), input/output data arguments (lines 45–47), processing elements requested (e.g., 3D grid in a GPU– line 50), task dependencies, and target device type (line 53). Moreover, applications can request the allocation of a temporary shared memory region available only for the duration of the kernel, which maps to the concept of local/shared memory found in other GPU programming APIs (CUDA, OpenCL).

Heterogeneous tasks are independent of the underlying target hardware, allowing a uniform expression of the application workflow whether they target CPUs, GPUs, or other device types. The computational kernel they represent is defined in a dialect similar to an OpenCL kernel (lines 3–21) that is translated appropriately for each target device. Input/output data arguments of a task are defined as the hetero_objects it needs to access, along with the access type required for each (read, write, read-write – lines 45–47). This information is used to issue the appropriate data transfers, maintain coherence, and infer task dependencies. Submitting a task for execution (line 56) does not immediately execute the respective kernel; instead, the runtime system enqueues the task execution request and immediately returns control to the user. The heterogeneous tasking framework provides methods to query the status of a kernel

```
// Device-independent kernel implementation
// device_global => Denote that data reside in device's global memory if it is a GPU
kernel(dgemm, (device_global double* A, device_global double* B, device_global double* C, long N),
{
   parallel_region(
      // Equivalents to CUDA blockIdx/blockSize/threadIdx
      int ROW = kernel_group_id_y * kernel_local_size_y + kernel_local_id_y;
      int COL = kernel_group_id_x * kernel_local_size_x + kernel_local_id_x;

      double local_sum = 0;

      if (ROW < N && COL < N)
      {
         for (int i = 0; i < N; i++)
         {
            local_sum += A[ROW * N + i] * B[i * N + COL];
         }

         C[ROW * N + COL] = local_sum;
      }
   )
})

int main()
{
   const int N = 1024;

   // Allocate NxN matrices A, B, C
   prema::hetero_object<double> m_A(N,N);
   prema::hetero_object<double> m_B(N,N);
   prema::hetero_object<double> m_C(N,N);

   // .request(false, true) => Request the framework to make the data available to the host, not to
       read (false) but to write(true), and return immediately
   // .get() wait until the data are available and get a pointer to them
   double *A = m_A.request(false, true).get();
   double *B = m_A.request(false, true).get();
   ...

   // Release data access from host
   m_A.release();
   m_B.release();

   {
      prema::hetero_task task;

      // Set the input/output matrices A, B, C
      task.arg(m_A).read(); //m_A is accessed as read-only
      task.arg(m_B).read(); //m_B is accessed as read-only
      task.arg(m_C).write().dim_x(); //m_C is accessed as write-only
                              //.dim_x() passes the size of dimension x to the kernel
                              //(the long N argument)

      // Set the thread dimensions
      task.set_threads({32, 32, 1}, {32, 32, 1});

      // Set the target device type
      task.device(device::GPU);

      // Set the kernel to execute
      task.submit(dgemm);
   }
}
```

FIGURE 3
An example of a DGEMM application using the tasking framework.

execution or wait for its completion. Moreover, task dependencies can be defined either explicitly by the user or implicitly by the runtime.

Applications can **explicitly define a task dependency graph** using the *add_dependency(task)* method of hetero_tasks. For example, one can create a graph where task *a* is independent

and tasks *b,c* depend on it by calling *b.add_dependency(a); c.add_dependency(a)*. Once their dependencies have been set, the application can submit the respective tasks for execution all at once. This approach allows the runtime system to improve performance while removing much of the burden of guaranteeing correctness from the application. To further reduce the effort required to guarantee correctness, the framework also supports **implicit task dependency detection** based on the arguments accessed by each hetero_task. Assuming that the application submits tasks in the correct sequential order, conflicting tasks are guaranteed to execute in the proper order; independent tasks will automatically explore maximum parallelism and avoid race conditions.

When explicit dependencies are used, the runtime simply stores the dependencies requested into the respective task struct. For implicit dependencies, the runtime has to automatically detect them based on the data accessed. Each time a new task is submitted to the framework, the hetero_objects passed as arguments are checked for read/write conflicts. For each hetero_object, the runtime maintains a list of the pending tasks that need to read from the hetero_object as well as the latest pending task that requires write access. If a write task arrives first, all following read tasks will add it as their dependence and will be blocked until it executes. If another write task arrives later, it will add all the read tasks as its dependencies and block until they all execute. In this scenario, the runtime is now free to remove the read tasks from its list since any new task (read or write) will now depend on the latest write task, which already depends on the read tasks about to be released.

3.1.3 Execution model

Heterogeneous tasks are executed asynchronously by the tasking framework. A task submitted for execution is appended to a list of task execution requests. The control is then immediately returned to the application, which can continue to issue more tasks or execute other work. A separate component of the runtime (optionally running in a separate thread) examines task execution requests and eventually schedules them for execution after performing the necessary steps to guarantee correctness.

The first step toward executing a task is to infer its dependencies with other tasks based on their data arguments. The runtime maintains a list of the currently submitted or running tasks that target each hetero_object; new tasks that access these hetero_objects with a conflicting access type have their dependencies set accordingly. Those with at least one incomplete dependence are pushed to another queue of blocked tasks; otherwise, they are appended directly to the scheduler's runnable work pool. Blocked tasks are periodically checked for resolved dependencies, and those with all their dependencies resolved are moved to the scheduler's runnable tasks pool. Each task maintains a list of other tasks that it depends on which have to complete before it can start its execution. Thus, for each task in the blocked list, the framework can check its list of dependencies directly. When a task in this list is detected to have completed, it is marked as complete in the dependency list, so that it is not checked again. Once all elements of its dependency list are marked as complete, the task is runnable and is removed from the blocked list.

Once all blocked tasks have been examined, the scheduler is ready to schedule the runnable tasks. At this stage, the scheduler decides the order in which the tasks should execute and the device where they should run based on the user's device type preference (i.e., the scheduler chooses the specific device ID while the user only gives a selection for a device type). The runtime will then reserve device resources and issue the data transfer requests of the input/output hetero_objects to be accessed on the chosen device. Moreover, it will automatically try to overlap the different operations, if possible, by utilizing the features provided by the target device's API (e.g., CUDA streams or OpenCL command queues). When all outstanding data transfers of a task have been completed, the computational kernel will be submitted to the target's work pool. Submitted tasks are periodically checked for completion by the runtime to update the status of pending dependent tasks. The lists maintaining the submitted, blocked, runnable, and running tasks are examined sequentially, and in order. In this way, the framework is guaranteed to correctly detect/resolve dependencies, update the data coherency-related structures, and move tasks among the lists. However, once tasks reach the runnable list, they are ensured that all of their dependencies have been resolved and, thus, the scheduler can run them in parallel and in any order.

3.1.4 Scheduler

With the introduction of more heterogeneous computing devices and workloads, it is expected that scheduling and load balancing will only become more complicated. To provide flexibility for different use cases, the actual implementation of the scheduler is designed to be modular and separate from the rest of the heterogeneous tasking framework. We provide the scheduler as an abstract class that only requires two operations to be implemented. The *push()* operation adds a new runnable task into the scheduler's work pool while the *pop()* operation returns the next task to be executed as well as the device it should run on. The abstract scheduler class allows the development of as simple or complex custom data structures and policies as the user might need.

3.1.5 Implementation

The heterogeneous tasking framework is implemented in the C++ programming language leveraging its performance and object-oriented design. It is developed in three software layers to allow easy integration with new device types, programming APIs, and scheduling policies (see Figure 4).

The **Device API** is the bottom layer, encapsulating the different operations provided by a heterogeneous device vendor. It consists of abstract C++ classes that expose virtual methods for operations required to (a)synchronously issue tasks and manage data in such devices, query their hardware specifications, and methods to retrieve the status of an asynchronous operation. Currently, we provide native support with CUDA and OpenCL for GPUs. The Device API provides the low-level, vendor-specific implementations of all abstractions of the tasking framework, like mapping hetero_objects to memory locations, hetero_tasks, and task execution requests to actual kernel invocations, memory transfer requests, etc.

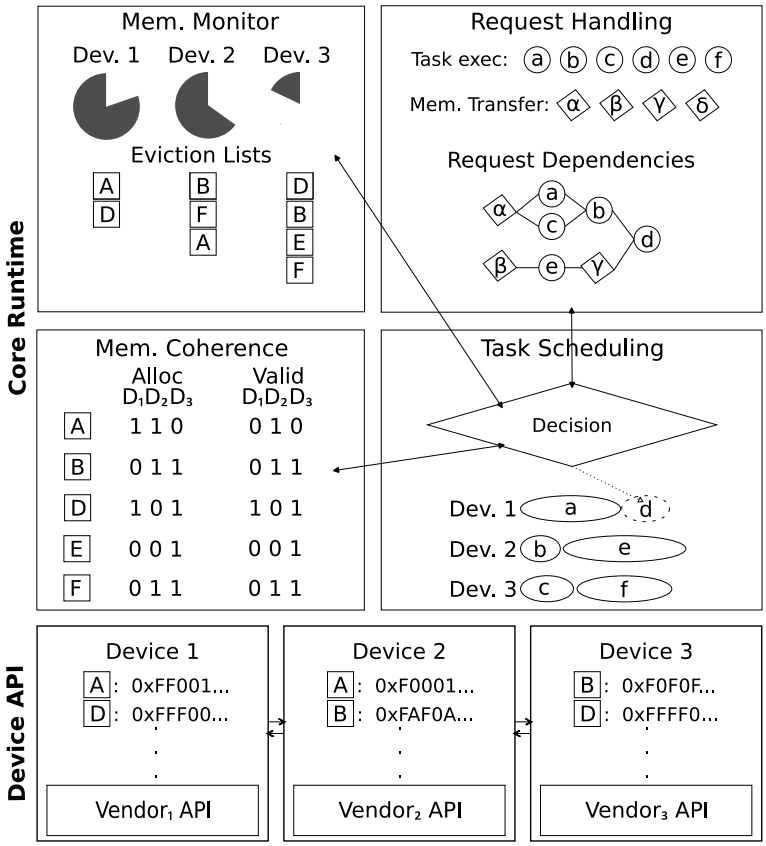

FIGURE 4
A high-level representation of the heterogeneous tasking framework software stack and operations. The operations performed by the Core Runtime include (a) Memory monitoring to keep track of the available device memory and deallocate unused objects when running low on resources, (b) Memory transfer and task execution request handling that dispatches such requests when it is safe, (c) Memory coherence among different copies of the same hetero_object in multiple devices, and (d) Task scheduling to optimize for a reduction in memory transfers and optimize overall execution time. The Device API exposes an abstract "device class" that encapsulates the implementation of different vendor interfaces uniformly. The device API maps high-level abstractions like hetero_objects and hetero_tasks to actual device-specific constructs.

The next layer is the **Core Runtime** layer, which provides the underlying implementation of the hetero_objects and hetero_tasks, monitors the coherence of the different copies of the data, and detects and enforces task dependencies. It utilizes the Device API to coordinate data transfers, guide the correct execution of tasks and signal the completion of different operations. This layer acts as the "glue" between the application preferences, the scheduler and load balancing policies, and the Device API.

At the top stands the **Application Layer**, which consists of a thin API that exposes the capabilities of the tasking framework in a high-level interface. In the current implementation, kernels are defined in a dialect similar to OpenCL through the use of macros which are expanded to implement a kernel version for each available target. We should note here that the kernel interface does not currently provide any automatic optimizations (e.g., shared memory, thread placement) and users should handle efficient kernel design by themselves. Our macros simply map the iteration space according to the target device or abstract device-specific operations/keywords with a uniform API. For example, a *parallel_region* block simply injects a barrier at its end for GPUs (since kernels run in parallel by default) while it generates a for loop, potentially parallelized to a set of tasks, for CPUs. Another example is the use of the keyword *shared* to denote an explicitly managed cache that maps to a *__shared__* memory in CUDA, *__local* memory in OpenCL or stack-allocated memory in CPUs. For each target, the kernel is compiled separately, using the vendor provided compiler and then it can be issued for execution using the vendor's library.

3.2 Heterogeneity within PREMA

Integrating heterogeneity in PREMA is a crucial requirement to handle the load of exascale-era machines. Applications should be able to use and transfer device memory in the context of remote handler executions without much hassle. A step toward this direction is to allow PREMA to send and receive buffers located in a GPU device either explicitly (currently CUDA only) or through the abstractions of the heterogeneous tasking framework we have introduced. The explicit approach allows users to utilize GPUs without confining them to use our heterogeneous tasking framework, facilitating interoperability with legacy CUDA codes. It also provides a barebone approach to integrate heterogeneity on top of the distributed system, which can act as the base case for our

performance evaluation omitting the overheads related to ensuring data consistency, dependency resolution, and task scheduling that the tasking framework adds.

3.2.1 Explicitly handling devices

In the explicit approach, the application can directly call the different GPU operations of the CUDA API to allocate/free memory, initiate transfers and execute kernels. PREMA provides a function to invoke remote handlers that include a GPU buffer as an argument; the function requests the ID of the remote process, the buffer to transfer, its size, the IDs of the source and target devices, and the handler (host function) to be invoked at the receiver. PREMA will transfer the buffer between the remote GPUs and invoke the handler when it has been completed. The handler can then invoke any GPU-related operation that targets this buffer safely. However, the application needs to guarantee that the handler does not return before the completion of the kernel since any buffers transferred through a handler will be freed at its return, including the GPU buffer. Waiting for all the device operations to complete [e.g., through *cudaDeviceSynchronize()*] is enough to guarantee correctness; however, this approach will harm PREMA's time-slicing abilities, preventing it from switching to other tasks while GPU operations are in progress. Thus, the user should follow a more complicated approach, querying the status of the operations without blocking (e.g., through *cudaEvents*) and periodically yielding control of the thread for PREMA to run background jobs.

3.2.2 Utilizing the heterogeneous tasking framework

To facilitate a higher-level interaction of PREMA with heterogeneous devices, we introduced a set of extensions allowing direct utilization of the abstractions provided by the heterogeneous tasking framework. Compared to the explicit remote handler invocation API, the user only needs to provide the handler to be executed, the target process ID, and the hetero_object passed as an argument (transferred).

Since the hetero_objects handle the location of the underlying data, the user does not need to specify their location. The framework automatically decides the device to store the received buffer on the target process. Once a hetero_object of a remote method invocation has been transferred, the designated handler is invoked on the target. The application can invoke tasks that utilize it on any available device type. Moreover, the application is guaranteed that any hetero_object that is the target of any hetero_task execution or messaging operation will live long enough for all such operations to complete, even if the handler returns earlier. In addition, the tasking framework will make sure that no other task can start executing on a hetero_object that is in the process of network transfer. A code example is shown in Figure 5 where a series of two distributed DGEMM invocations is implemented in heterogeneous PREMA. Two mobile objects create matrices A, B, and C, then create mobile pointers out of their data and exchange them with each other. Next, each mobile object invokes the first DGEMM on itself and sends the result to the remote object invoking the second DGEMM. Note that there is no need to explicitly handle the data buffers for the network transfer (lines 29, 55). Also, the application does not need to explicitly wait for the completion of the DGEMM task (line 27) before sending the results to the remote mobile object (line 29). Finally, even though the result of the first DGEMM is stored in a local variable (lines 20, 24) and the handler can return before the asynchronous send has completed, the data will be transferred correctly. PREMA and the tasking framework will make sure that data are consistent and updated in the correct order.

Another desirable requirement provided through the hetero_objects on top of PREMA is the ability to "put" and "get" data between potentially distributed devices. A new extension allows users to create global pointers for hetero_objects, i.e., unique identifiers referenceable from all processes in the distributed system. When an application needs to store/retrieve data to/from a remote hetero_object, it just needs to provide its global pointer and the location (hetero_object or pointer) of the data to be read/written, along with a callback that is triggered on the target, signaling the completion of the operation.

3.2.3 Implementation

Depending on the capabilities of the underlying communication library and the target device hardware, the actual implementation of the memory transfers differs to leverage heterogeneity-aware communication substrates. When the application utilizes hetero_objects, and the communication substrate is not heterogeneity-aware, the implementation of the memory transfers includes the following steps (also see Figure 6).

PREMA will automatically request an asynchronous read of the hetero_object data from the device to the host. The tasking framework will guarantee that the device-to-host transfer will start once all previously submitted, conflicting tasks have finished and prevent any new ones from running before PREMA has finished its network transfer. Next, a header message encapsulating the handler's metadata (e.g., data pointer, target, etc.) is prepared and pushed to a queue of pending send message requests. The header will also incorporate a future returned by the previous step that allows for checking for the transfer's progress. The outgoing message requests queue is checked periodically by PREMA. When the future of a message signifies the completion of a device-to-host transfer, the message is ready to be sent through the network. PREMA will asynchronously send two messages, one for the metadata and one for the hetero_object data, utilizing the MPI as the communication substrate. Finally, when the transmission of the two messages has been completed, PREMA will release its read request from the hetero_object, notifying the tasking framework that the object can be safely modified or deleted by another task.

On the receiving side, once the first metadata message is detected and received, the information it carries is used to allocate the required memory to store the actual data of the hetero_object. Next, the second message with the actual hetero_object data is received in the newly allocated buffer. A request to allocate the received data in a device is sent to the tasking framework, and the metadata message along with this request is enqueued into a pending handler execution request queue. Finally, the scheduler will pick one of the pending handler execution requests and run

```
1
2  // PREMA mobile object handler implements D = A * temp
3  DEFINE_MP_HANDLER(execute_second_dgemm_handler)
4  {
5     // Get handle to the hetero object received from line 30
6     prema::hetero_object<double> m_B = get_hetero_object<double>();
7     prema::hetero_task task;
8
9     // Set the input/output matrices
10    task.arg(this->m_A).read();
11    task.arg(m_B).read();
12    task.arg(this->m_C).write().dim_x();
13    task.set_threads({32, 32, 1}, {32, 32, 1});
14    task.device(device::GPU);
15    task.submit(dgemm);
16 }
17 // PREMA mobile object handler implements temp = B * C
18 DEFINE_MP_HANDLER(execute_first_dgemm_handler)
19 {
20    prema::hetero_task task;
21    hetero_object m_C;
22    task.arg(this->m_A).read();
23    task.arg(this->m_B).read();
24    task.arg(m_C).write().dim_x();
25    task.set_threads({32, 32, 1}, {32, 32, 1});
26    task.device(device::GPU);
27    task.submit(dgemm);
28    // Invoke execute_second_dgemm_handler to other_mp passing matrix m_C as input
29    prema::mp_send(this->other_mp, execute_second_dgemm_handler, m_C);
30 }
31
32 struct mobile_object_data
33 {
34    prema::hetero_object<double> m_A;
35    prema::hetero_object<double> m_B;
36    prema::hetero_object<double> m_C;
37 }
38 // Implements D = A*B*C
39 int main()
40 {
41    prema::init();
42    const int N = 1024;
43
44    // Allocate NxN matrices A, B, C
45    prema::hetero_object<double> m_A(N,N);
46    prema::hetero_object<double> m_B(N,N);
47    prema::hetero_object<double> m_C(N,N);
48
49    // Populate A, B
50    ...
51    // mobile_object_data => A struct that encapsulates the computation data
52    mobile_object_data my_data(m_A, m_B, m_C);
53
54    // my_mp is a reference to the (currently) local mobile object
55    prema::mobile_ptr my_mp(my_data);
56
57    // other_mp is a reference to the remote mobile object
58    prema::mobile_ptr other_mp = /*get remote mobile_ptr*/;
59    my_data.other_mp = other_mp;
60
61    // Invoke handler execute_first_dgemm_handler on mobile object referenced by my_mp
62    prema::mp_send(my_mp, execute_first_dgemm_handler);
63    prema::shutdown();
64 }
```

FIGURE 5
An example of a series of two DGEMM invocations where the results of the first are needed on a remote node to invoke the second, using the heterogeneity-aware PREMA.

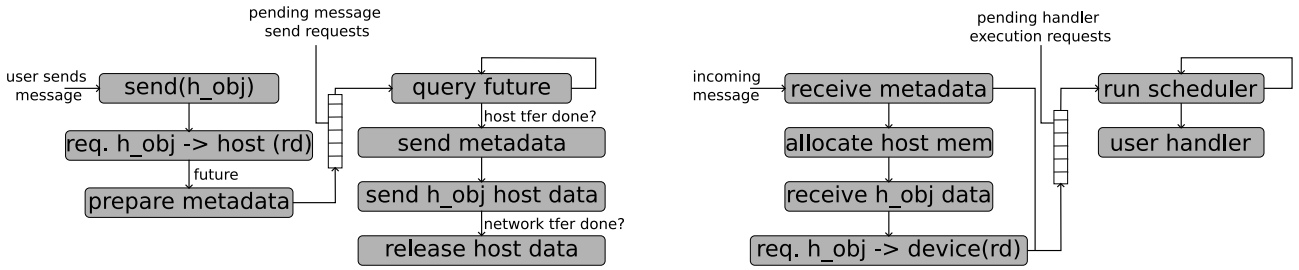

FIGURE 6
Heterogeneous data send **(left)** and receive **(right)** in PREMA with no GPU-aware interconnect.

the respective user handler. Note that in this case, PREMA does not wait for the host-to-device transfer to complete before starting the handler, allowing code independent of the hetero_object itself to run, overlapping the transfer. Furthermore, since the data of the hetero_object will be used through a hetero_task, the tasking framework will ensure that any task execution will be delayed until the transfer has been completed.

The "put" operation is almost identical, with the small difference that the receiver does not need to allocate new host memory. Since the hetero_object already exists on the target, PREMA will request write access on the host side, and once access is granted, it will receive the data directly in the hetero_object's host memory. Note that no transfer from device to host is performed since the data will be overwritten from the receive. The request will just guarantee that the network transfer will not conflict with any other task running on this hetero_object. A "get" operation also utilizes the "put" operation internally; the initiator/reader invokes a handler on the mobile object owning the data that calls the "put" operation and stores the requested data into the initiator's desired hetero_object.

The host-staging step can be skipped if the communication library/hardware and compute devices support direct transfers between distributed devices (currently only tested for CUDA-OpenMPI). In this case, the hardware can perform the data movement between the GPU devices directly through the network interface card (NIC), avoiding the intermediate transfers to the main memory. Thus, in an implementation for the case where the user uses hetero_objects, the process is as follows (also see Figure 7).

PREMA will automatically asynchronously request read access to the hetero_objects pointer in the current device. Again, the tasking framework will ensure that no conflicts with active tasks will be possible, but no transfer will occur. Same as the non-GPU-aware case, a metadata message is prepared that includes a future generated from the previous step's request and is appended to the message send requests. Once it is safe for PREMA to access the device memory, the message is ready to be sent through the network. The metadata message is sent first, followed by the data in the device. Under specific conditions, some MPI implementations can directly target CUDA device memory which is used in this case. When the message transmissions have been completed, PREMA will release its read access to the hetero_object. This will allow other task/messaging operations to modify the hetero_object safely.

On the receiving side, once the metadata message is received, the tasking framework is requested to create a dummy hetero_object on the device that can accommodate the incoming data. Then the metadata message along with the dummy hetero_object are enqueued to the pending handlers queue for execution. When the hetero_object allocation is complete, PREMA will receive the actual data directly in the memory allocated in the device and notify the tasking framework that it no longer needs to keep access to the device pointer. Finally, the scheduler will pick one of the pending handler execution requests and run it. Again, the handler can start before the actual data have been received in the hetero_object, leaving the tasking framework to guarantee that no conflicts will occur by delaying tasks as needed.

Like in the previous case, the "put" operation is similar, with the only difference that the receiver can directly receive incoming data on the current location of the hetero_object. Since the hetero_object already exists on the target, PREMA will request write access on its current device location. Once access is granted, it will receive the data directly in the hetero_object's device memory. Again, data will be overwritten from the receive; thus, the request will guarantee that the network transfer will not conflict with any other task running on this hetero_object.

4 Performance evaluation

This section investigates optimizations that can improve the performance of different operations provided by the heterogeneity-aware PREMA and the tasking framework and presents the performance of the optimized implementation on a proxy application. We used a small 16-node cluster, with each node consisting of two Intel Xeon Gold 6130 20-core CPUs (2.1 GHz) and four NVIDIA Tesla V100 GPUs. The connection between the host and the GPUs is a single PCIe buss with a bandwidth of 10 GB/s while the network interconnect is a Mellanox Infiniband EDR backbone providing 100Gbps throughput. For all of the benchmarks presented below, the heterogeneous tasking framework uses a locality-aware scheduler, scheduling tasks to devices that already have a copy of (some of) the needed data (e.g., from a previous task execution). Moreover, it will try to map data of the mobile object to the same GPU as they would most likely be dependent on each other.

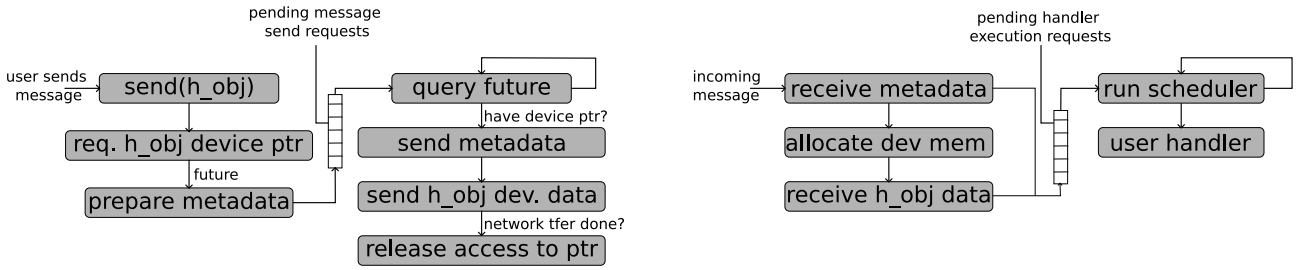

FIGURE 7
Heterogeneous data send **(left)** and receive **(right)** in PREMA with GPU-aware interconnect.

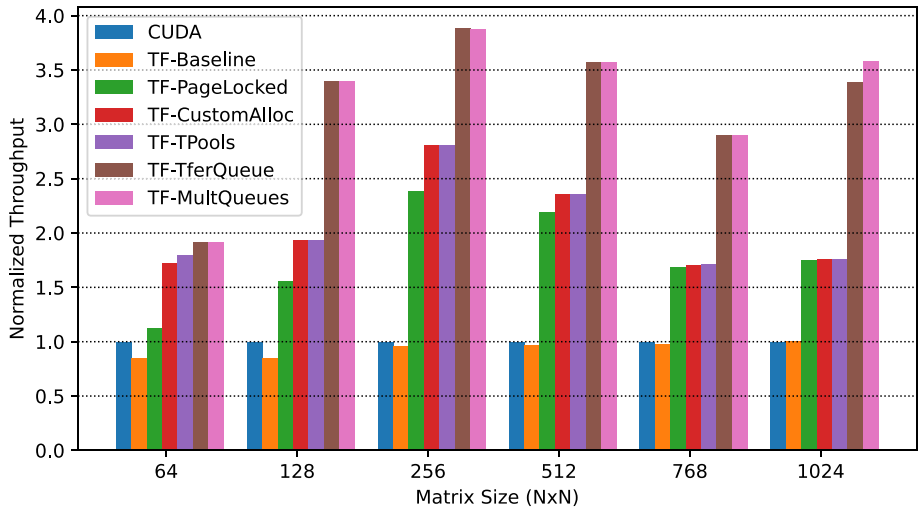

FIGURE 8
Effects of incrementally applying different optimizations on the heterogeneous tasking framework's performance for a matrix multiply benchmark on a single GPU. The performance improvements are evaluated on multiple matrix sizes and normalized by a CUDA implementation's performance. **CUDA**: The baseline CUDA implementation. **TF-Baseline**: The tasking framework (TF) without any optimizations. **TF-Pagelocked**: TF utilizing Page-locked host memory. **TF-CustomAlloc**: TF after applying the custom memory allocator optimization. **TF-TPools**: TF utilizing request pools. **TF-TferQueue**: TF after introducing a separate queue for memory transfers. **TF-MultQueue**: Final version of TF, utilizing multiple queues for kernel invocations.

4.1 Heterogeneous tasking framework

We evaluate different performance optimization techniques on this framework for NVIDIA GPUs using a simple, double precision matrix-matrix multiply benchmark and compare the throughput achieved in 100 iterations with a pure CUDA implementation. In each iteration, the three matrices are allocated in the device, input data are transferred, and the compute kernel is executed. Note that the results are not copied back to the host to prevent the pure CUDA implementation from blocking. We incrementally apply optimizations and show their effect. As shown in Figure 8, the baseline bar indicates that the framework adds some overhead on top of the naive CUDA implementation, which is significant, especially for smaller matrices. This overhead stems from the added value the tasking framework provides related to maintaining ordering, dependencies and guaranteeing data coherency among different devices. PREMA overcomes some of these overheads with the optimizations presented. Note that many optimizations applied automatically by the heterogeneous framework could also have been implemented directly in the CUDA implementation. However, it would require much more effort from the user and a relatively more extensive and more complex application code. Moreover, the code would need to be rewritten to run on non-NVIDIA devices.

4.1.1 Page-locked host memory

To fully utilize the bandwidth capabilities of the respective hardware, NVIDIA GPUs require that host data to be transferred to the GPU should reside in a page-locked memory region. Moreover, this is the only way that device-to-host transfers can be asynchronous with respect to the host. Thus, applications need to explicitly (de)allocate memory in a unique way, which increases code complexity and induces an overhead much higher compared to regular memory allocations. We incorporated this optimization into the framework to relieve applications and PREMA from explicitly handling this burden. For this purpose, the framework allocates a large chunk of page-locked memory at the initialization

step that is later used as a memory pool for host memory allocations and prevents further expensive requests for page-locked host memory allocations.

The optimization gives a significant boost that ranges between 30% and 145% for different matrix sizes (Figure 8; TF-PageLocked). Specifically, the smallest improvement is observed in smaller matrices (64 × 64) with 30%, followed by the larger ones (768 × 768 and 1024 × 1024) with about 70%. In the 128x128 case, it attains 90%, while the most notable improvement with more than 100% is achieved when 256 × 256 (120%) and 512 × 512 matrices (145%) are used.

4.1.2 Custom device memory pools

Allocating and freeing device memory are expensive operations that may require synchronization between the host and the device. Moreover, the two functions might require the completion of previously issued asynchronous operations before running. The proposed runtime system uses a custom memory allocator per device to avoid overheads from constantly requesting new memory (de)allocations. During the initialization of the runtime system, a request to allocate most of the available memory of each device (except the host) is issued, and the custom memory allocator handles the returned memory. When memory needs to be allocated, the custom allocator is used instead of the one provided by the device library.

The most significant improvement of this optimization is manifested for the smallest matrix case (64 × 64) with another 60%. As the size of the matrices increases, the improvement observed declines with 25%, 12%, and 6%, respectively, for matrices ranging from 128 × 128 to 512 × 512. For larger matrices, the improvement is negligible (Figure 8; TF-CustomAlloc).

4.1.3 Enabling concurrent GPU operations

So far, all the optimizations implemented were focused on improving the overlap of the CPU and GPU operations; however, processes that run in the GPU are still serialized by default. We enable implicit concurrency between the different GPU operations by utilizing multiple execution streams provided by the device implementations (OpenCL command queues or CUDA streams for NVIDIA GPUs). Our implementation uses multiple streams for submitting computation kernels and two for memory transfer operations (one for each direction). We found that five streams for computation kernels are generally enough to saturate the device capabilities for concurrent kernel executions. Still, we also provide an environmental variable that allows the application to change this without recompilation.

The results of this optimization are substantial in most matrix sizes when the overlap between memory transfers and kernel executions can be large enough. This is the case for all matrix sizes larger than 64 × 64. Since the overhead of transferring data to the device is substantial, a great percentage of that can be overlapped with the kernel execution of the previous iteration. Thus, the results show improvements of 75%, 50%, 50%, 85%, and 100%, respectively, for matrices of size 128 × 128–1024 × 1024. On the other hand, the improvement in the case of 64 × 64 is 10% due to the small amount of data that needs to be transferred (Figure 8; TF-TferQueue, TF-MultQueues). The final optimization applied in this aspect was to use more than one stream per data transfer direction. However, this did not improve performance further because the device hardware only supports two copy engines.

4.1.4 Other optimizations

We have introduced request memory pools to mitigate the effects of system calls and thread synchronizations. Request pools are maintained per active thread, and the memory of a request is recycled in the pool once the respective operations have been completed. Another optimization regarding requests was implemented on the queues used to submit a request to a device. Initially, we used structures provided by the C++ STL, protected by a mutex, to implement such queues. These queues were substituted with custom lists that avoid allocating nodes to store new elements. Moreover, the mutex locking step was moved after the queue's size was checked to eliminate unneeded locking operations. This optimization is less important than the ones discussed above, about 2% improvement, but helps to attain a more consistent latency, especially when the dedicated thread is used (Figure 8; TF-Tpools).

4.1.5 Putting it all together

To summarize, we have applied a series of optimizations that affected the performance of the tasking framework on the specific benchmark in different percentages depending on the size of the matrices. The most important optimizations include the automatic use of page-locked host memory and the introduction of multiple streams. However, the custom device memory pool also substantially improved the cases of smaller matrices. The overall performance improvement achieved from this series of optimizations can exceed 300%, depending on the size of the matrices. Our framework offers all these optimizations with minimal user involvement and will continue to improve without any modifications required in the application code. Note that the performance improvements will increase as the number of memory transfers per computation decreases.

4.1.6 Multi-device platforms

The above results show the performance of the heterogeneous tasking framework on a single device. However, our framework is able to utilize multiple devices automatically. This is where the importance of using dedicated threads per device becomes apparent. While dedicated threads do not help when only a single GPU is in use (as shown in Figure 9), they are crucial to scale beyond a single device since they allow the host side to provide enough work for them to process and saturate their capabilities. Figure 9 shows that the framework can scale almost linearly as we add more devices, achieving up to 3.8 speedup on 4 GPUs. The superlinear speedup observed in the case of one and two GPU devices is an effect of the optimizations presented so far for a single device. As we add more devices, the effect of these optimizations declines, and the speedup becomes linear. The green line presents the performance improvement stemming only from the introduction of dedicated threads, which shows increasing gains

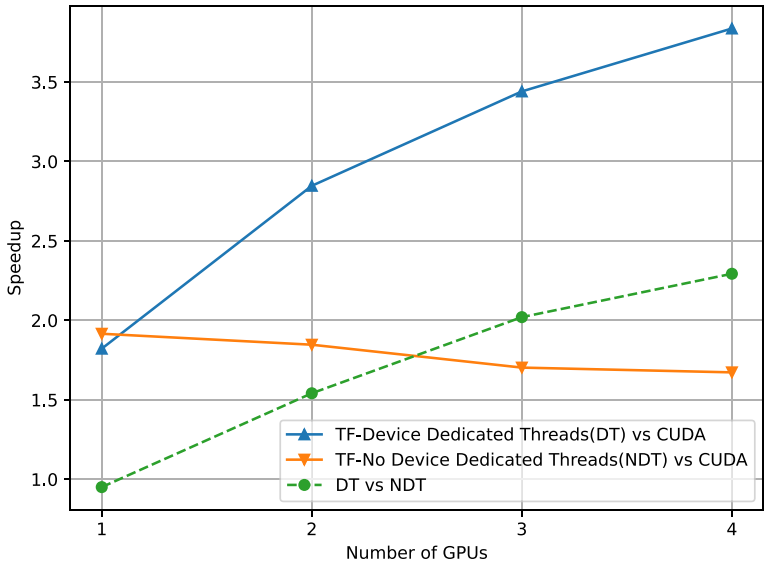

FIGURE 9
Performance of the heterogeneous tasking framework on a 64 × 64 matrix multiply benchmark utilizing multiple GPU devices with and without dedicated threads per device. The performance is shown as speedup against a simple CUDA implementation. The performance gain between the two approaches is also presented.

as we introduce more GPUs to the system. This is expected as a single CPU cannot saturate the capabilities of mulitple GPUs. We must note that the hardware available to us constraints the framework's capabilities. All the devices in the specific machine share a single bus with the host. Thus, the maximum memory transfer throughput is constant whether one or four devices are utilized. For the specific benchmark, a maximum of 2 64 × 64 matrices can be moved simultaneously to any device since they are enough to saturate the available memory bandwidth between the host and the devices. Given a more capable hardware, we expect the framework to perform much better in a multi-device context.

4.2 Heterogeneous PREMA

To optimize the performance of the new heterogeneity-aware version of PREMA, we experiment with optimizations that can help us mitigate the overheads following the implementation of remote handler invocations that include heterogeneous memory both without and with the tasking framework (without a dedicated thread). We evaluate our optimizations on a simple ping-pong benchmark for inter-node communications and compare it with an MPI+CUDA implementation. The benchmark runs 100 ping-pong iterations with message sizes ranging between 8 bytes to 8 MBs, and the average latency and bandwidth observed per message size are reported.

4.2.1 Device message receiving cache

In PREMA, messages are one-sided and asynchronous and are received implicitly to invoke a designated task on their target. Thus, the receiver cannot specify a memory region where the message buffer shall be stored (like MPI). PREMA has to dynamically allocate memory to receive the incoming message buffer and provide it to the application handler invocation. In our initial implementation, without the tasking framework, simply allocating new device memory for each incoming message resulted in poor performance, increasing the latency experienced up to ten times compared to the respective MPI implementation. We avoided this behavior by allocating a cache in the device specifically for the buffers of received messages. When a new message buffer is about to be received, memory is requested from the cache instead of the device API if possible. The cache allowed us to attain performance within 10% overhead of that achieved by the MPI (Figure 10; PREMA+CUDA). It is also important to note here the consistent "spike" observed in the MPI performance for messages of size 256B, which does not seem to affect our implementation when using the device cache. It is not clear what is the reason causing the spike in the MPI implementation, but it seems to be an effect of caching in the CUDA architecture. This would explain why introducing our custom cache alleviates this behavior since a specific "chunk" of memory is always reused, allowing the architecture to cache it. On the other hand, the MPI+CUDA implementation needs to allocate a new "chunk" for each new message size that potentially points to a different address in memory which needs to be cached again.

4.2.2 Preallocating hetero_objects

Hetero_objects automatically utilize memory pools for device memory, thus, implicitly overcoming the issue faced in the case where device memory is handled explicitly and achieving performance within 25% of the MPI+CUDA implementation (Figure 10; PREMA+TF-0). However, we can still improve some latencies caused by the constant allocation and deallocation of temporary hetero_objects that wrap message buffers targeting device memory. Specifically, we found that the data structures

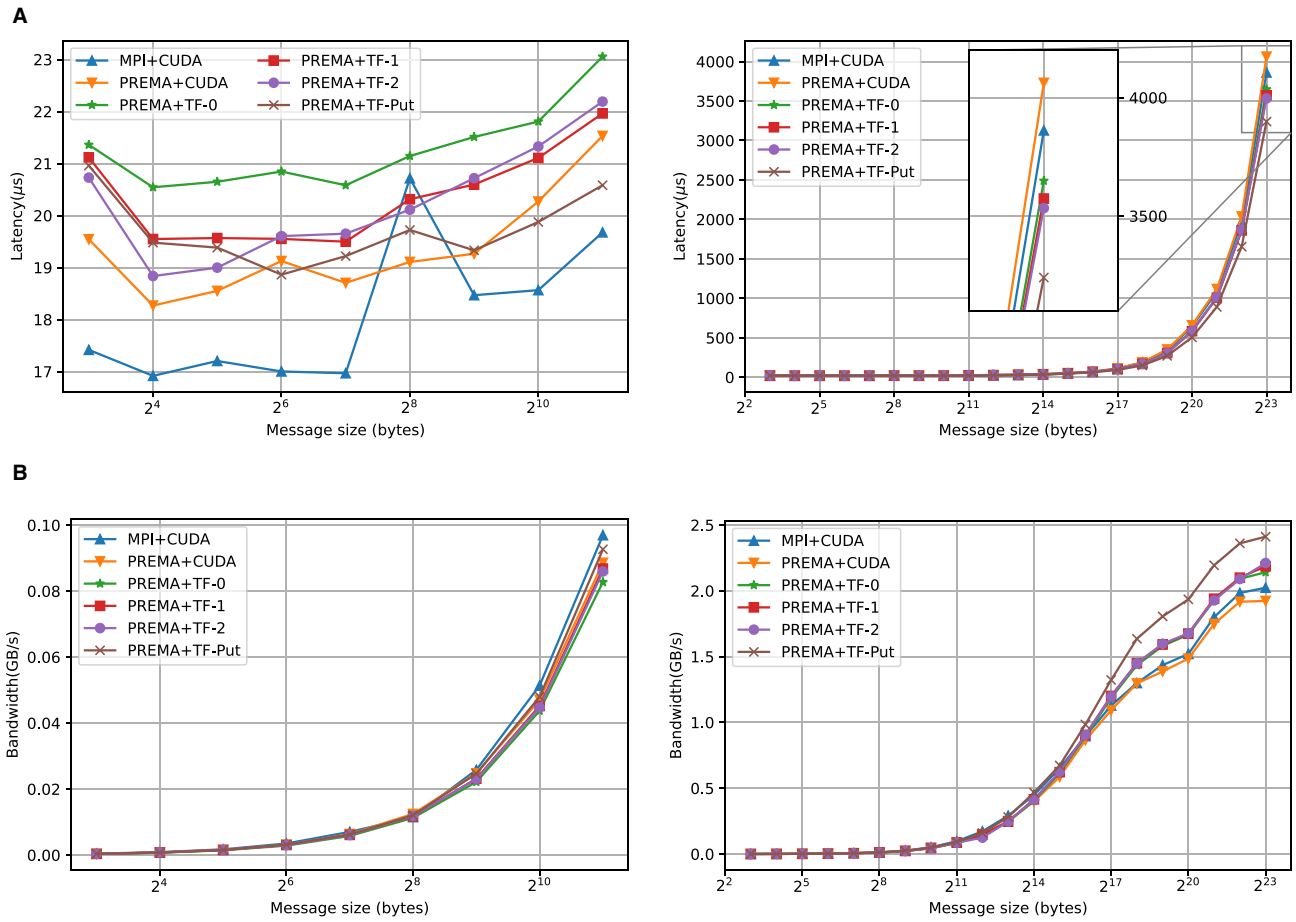

FIGURE 10
Performance evaluation of different optimization techniques in terms of **(A)** latency and **(B)** bandwidth on a ping-pong benchmark without direct network device-to-device transfers. Optimizations are compared against the baseline performance of an MPI+CUDA implementation. **PREMA+CUDA**: Integration of PREMA with CUDA directly, without involving the heterogeneous tasking framework. **PREMA+TF-0**: The baseline integration of PREMA and the heterogeneous tasking framework. **PREMA+TF-1**: PREMA+TF-0 plus the introduction of memory pools. **PREMA+TF-2**: PREMA+TF-1 plus the optimization of avoiding message buffer copies for very small messages. **PREMA+TF-Put**: The remote put operation after the optimizations of PREMA+TF-2. **(A)** Latency for small messages (left) and overall (right). **(B)** Bandwidth for small messages (left) and overall (right).

allocated for a hetero_object for bookkeeping different operations targeting the object in various devices can significantly affect communication performance. Since these structures can be allocated in advance, we use a pool of preallocated semi-initialized hetero_objects to mitigate this effect. This optimization improved the latency experienced for small messages by about 10%, bringing the performance of PREMA within 15% overhead of MPI+CUDA, as can be seen in Figure 10 (PREMA-TF-1). Moreover, for larger messages the performance achieved is better from the MPI+CUDA by almost 10% as can be seen when zooming in the latency of 8MB messages.

4.2.3 Avoiding message buffer copies

The process of inter-node message transfers presented in Section 3.2.3 with host-staging includes an optimization for small buffer sizes to only send one message. Small messages with a size up to 512 bytes (header + buffer size) will consist of the actual application data/buffer appended at the end of the message. This helps optimize the performance of small messages where even negligible overheads are noticeable. However, when hetero_objects are used, this introduces an extra copy. As explained before, requesting access to the underlying data of a hetero_object will implicitly copy the data to the host in a buffer maintained by the framework. To append the data at the end of the message header, PREMA needed to copy the data from the framework's location to the message header.

A new method is introduced in the hetero_object API that allows the user to request a copy of its underlying data to a designated memory region at the host. PREMA utilizes this feature to request from the framework to directly transfer the device data at the end of the message header buffer. This change slightly improves the communication critical path and attains up to 5% lower latency for smaller messages (Figure 10; PREMA-TF-2).

4.2.4 Put operation

Another operation introduced to further increase the performance of inter-device communication over the network is the put operation. As mentioned earlier, the put operation allows

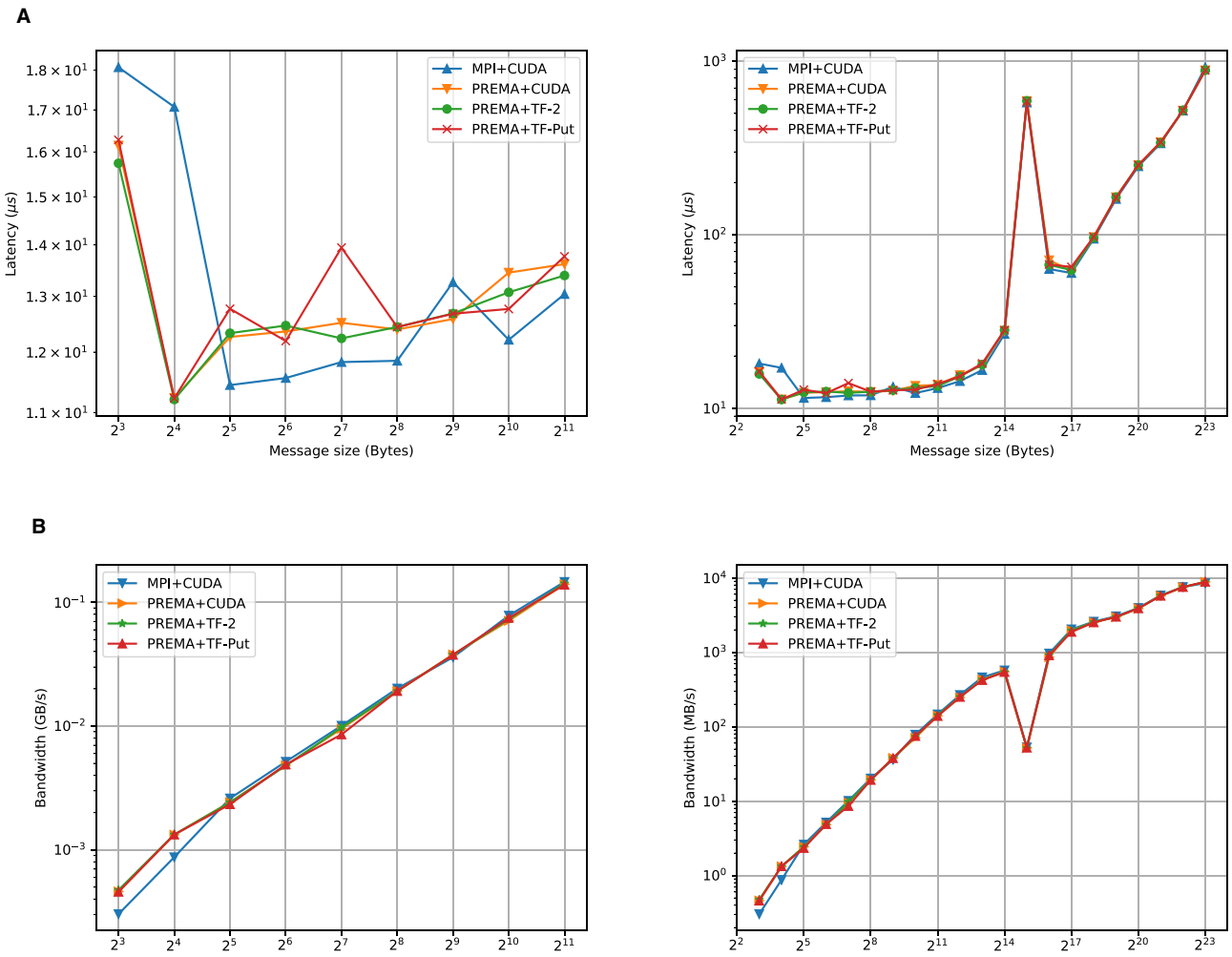

FIGURE 11
Performance evaluation of different optimization techniques in terms of **(A)** latency and **(B)** bandwidth on a ping-pong benchmark with direct network device-to-device transfers. Optimizations are compared against the baseline performance of an MPI+CUDA implementation. **PREMA+CUDA**: Integration of PREMA with CUDA directly, without involving the heterogeneous tasking framework. **PREMA+TF-2**: Integration of PREMA and the heterogeneous tasking framework incorporating all the optimizations presented in this section. **PREMA+TF-Put**: The remote put operation after the optimizations of PREMA+TF-2. **(A)** Latency for small messages (left) and overall (right). **(B)** Bandwidth for small messages (left) and overall (right).

PREMA to utilize the existing memory of heterogeneous objects that is also page-locked. By leveraging these optimizations, the put operation outperforms all previous optimizations that use the tasking framework since the transfer from the host to the device is much faster on the receiver side. Figure 10 (PREMA-TF-Put) shows its performance reaching and overcoming the one of the PREMA+CUDA implementation for messages larger than 64 bytes and even the MPI+CUDA for messages larger than 8KB achieving up to 20% better performance for messages of 8MB.

4.2.5 Direct to device transfers

The optimizations presented so far target the generic implementation of heterogeneity on top of PREMA, where the communication library/hardware is not heterogeneity-aware. However, as mentioned in detail in Section 3.2.3, PREMA can leverage the capabilities of hardware/libraries that have been integrated with support for direct device-to-device communication. Following the previously presented procedure, the latencies observed for heterogeneity-aware hardware can be significantly mitigated. Figure 11 shows the performance of the optimized version of each operation and the MPI+CUDA when direct-to-device communication is possible. The attained performance, in this case, is up to 100% better than the host-staging case for small messages and up to 200% for large messages, as shown in Figure 12. In this case, we experience a "spike" across all implementations for 32KB-sized messages. Even though we do not know the exact details of this issue, it seems related to the protocol switch limits that OpenMPI uses. Specifically, OpenMPI switches to "pipelined transfers of size 128 KB through host memory" for message sizes $\geq$ 30,000 bytes (OpenMPI, 2024). Thus, one explanation for this spike could be the cost of the host buffer allocation/initialization/registration, which is alleviated for the following messages by reusing the buffer.

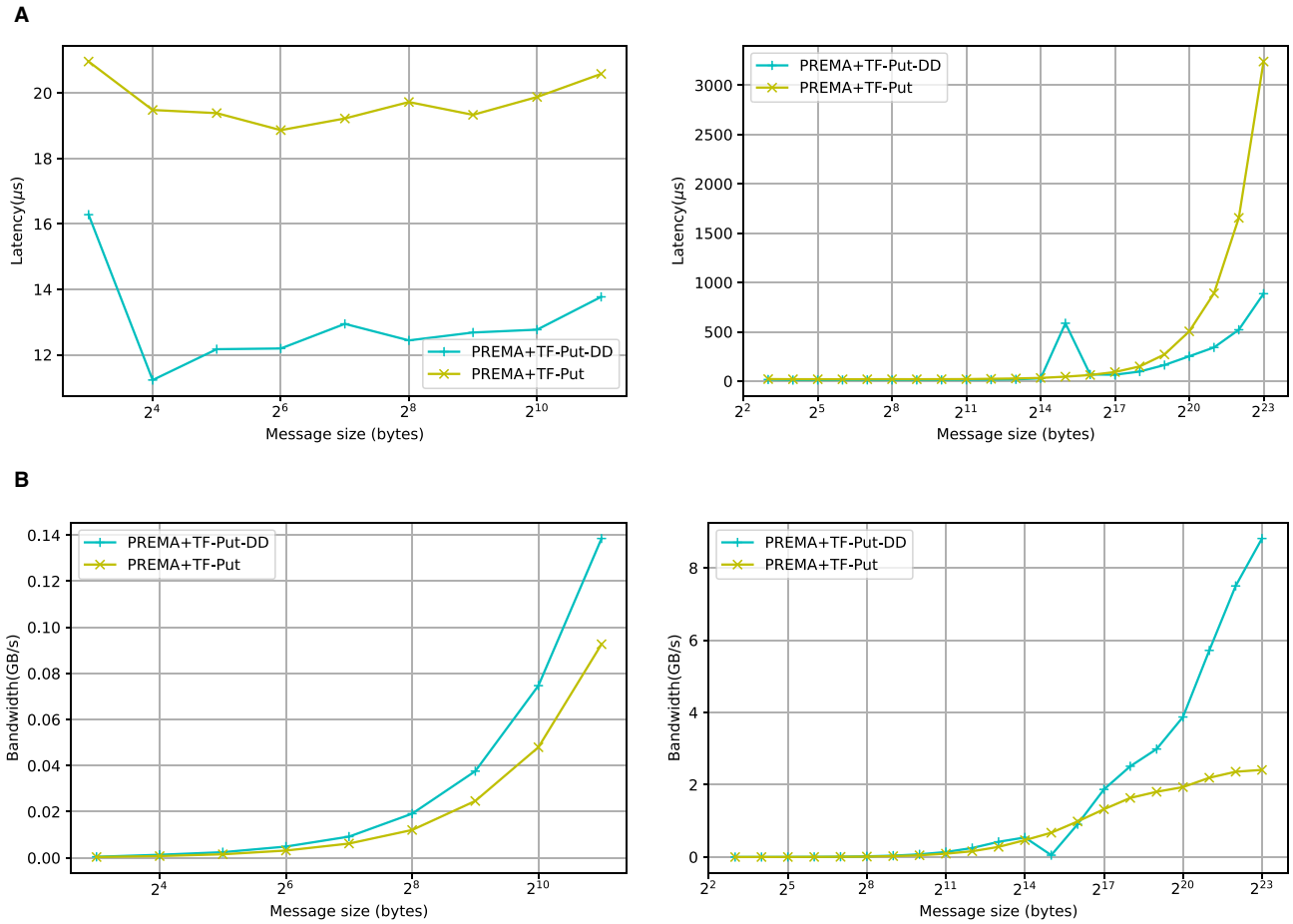

FIGURE 12
Comparison of the optimal performance in terms of **(A)** latency and **(B)** bandwidth using host-staging or Device-to-Device (DD) network transfers for the ping-pong benchmark. **(A)** Latency for small messages (left) and overall (right). **(B)** Bandwidth for small messages (left) and overall (right).

4.2.6 Summary

Overall, all three operations introduced in PREMA to handle heterogeneity, including the transfer of CUDA buffers, hetero_objects, and the remote put operation, significantly simplify application development, saving hundreds of lines of boilerplate code while maintaining reasonable overhead (10%–15%) compared to MPI+CUDA regarding latency and bandwidth. Moreover, it is interesting to notice that some of these operations, like the put operation, outperform MPI+CUDA (by up to 20%) for large messages (Figures 10, 11) by implicitly leveraging from the page-locked memory of hetero_objects. Page-locked memory forces the operating system to lock a virtual memory address to a specific physical address, allowing CUDA GPUs to use DMA and significantly improve the throughput of read and write operations.

4.3 Proxy application: Jacobi3D

To evaluate the performance of the distributed memory framework, we have adapted a proxy Jacobi3D iterative method application (Laboratory-UIUC, 2022) on PREMA. The Jacobi3D iteration operates on a 3D grid where each update is a stencil computation that calculates a weighted average of a cell with its neighbors in the six cardinal directions, as in Equation (1):

$$C_{x,y,z} = \alpha C_{x,y,z} + \beta(C_{x-1,y,z} + C_{x+1,y,z} + C_{x,y-1,z} + C_{x,y+1,z} + C_{x,y,z-1} + C_{x,y,z+1}) \quad (1)$$

where C_{xyz} is the cell in position (x,y,z) of the grid.

The proxy performs a fixed number of iterations of the Jacobi iterative method on GPUs in a 3D domain of cells decomposed into cuboids and wrapped into mobile objects, where each mobile object consists of approximately equal number of cells. In each iteration, the mobile objects exchange halo data, packing the GPU data and transferring them to their neighbors (up to 6 neighbor cuboids; two for each dimension). On the receiving side, the data are unpacked into the GPU, and once all halos have been received, the Jacobi update is executed. Figure 13A shows the execution time of the heterogeneous PREMA vs. the MPI+CUDA counterpart for a 1024 × 1024 × 768 domain (strong scaling). The implementation with PREMA achieves up to 23% better performance. For weak scaling, the domain's size is increased according to the number of distributed GPUs. The

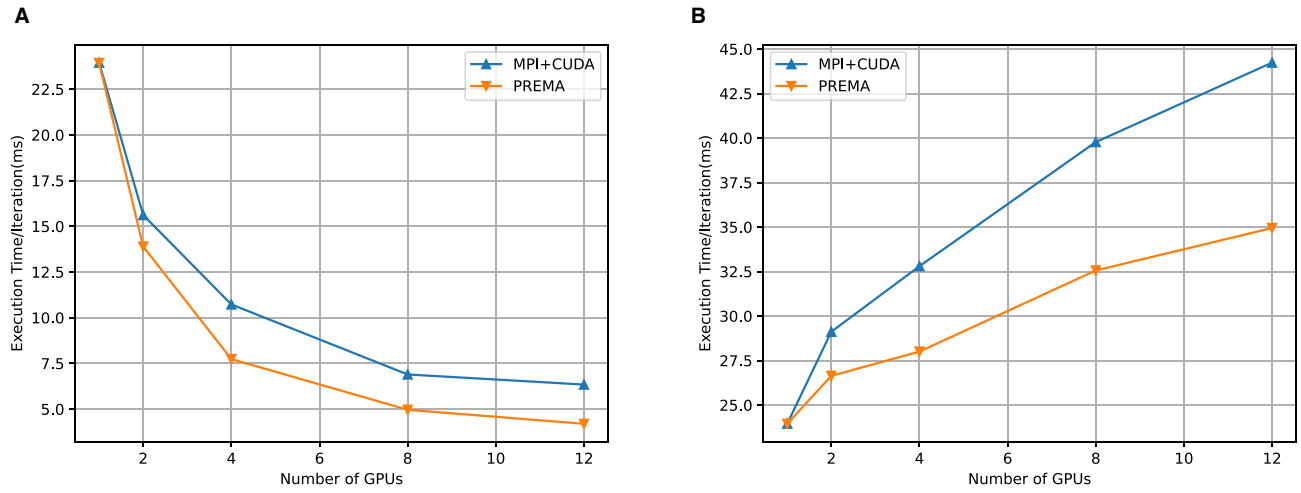

FIGURE 13
Strong **(A)** and weak **(B)** scaling performance of the Jacboid3D proxy application.

performance achieved is up to 25% (Figure 13B) better than the MPI+CUDA implementation. The improvements observed stem from automatically overlapping message passing, host-device memory transfers, and kernel invocations.

4.4 Over-decomposition

A common practice that PREMA applications utilize for performance improvement is over-decomposition. Over-decomposition is used to decompose the data domain into more chunks than the number of PEs, allowing PREMA more flexibility to load balance workload and overlap latencies. The effectiveness of this approach has already been demonstrated in previous work for homogeneous platforms (Thomadakis et al., 2022; Thomadakis and Chrisochoides, 2023). In the context of heterogeneity, host-to-device, and device-to-host memory transfers are broken into pipelined pieces and can be overlapped much more easily with the following kernel invocations. An example of the execution timeline that is achieved can be seen in Figure 14. Please note that the duration assigned to each task in the example is not strictly accurate. For instance, the time allocated for processing the halos is not necessarily equal to the time spent in the computation process. However, such exaggeration is employed for the sake of clarity and readability, rather than precision. The effects of over-decomposition are shown in Figure 15 for the same Jacobi3D proxy application. Different levels of over-decomposition are attempted in this benchmark, with each level up attaining improvements over the MPI implementation as well as the PREMA implementation without over-decomposition (OD1) when using more than 1 GPU. The best performance is observed with an over-decomposition of two, which achieves improvements of up to 40% vs. the MPI implementation and about 20% over the initial PREMA implementation for 12 GPUs. When only a single GPU is in use over-decomposition does not seem to help and can actually harm performance (e.g., OD6). The reason for this decline is that in a single GPU and over-decomposition level 1 the data will only need to be moved in once and no packing/unpacking needs to take place, since they are no halo buffers to exchange. By increasing the level of over-decomposition, we create the need for halo buffer exchanges among different addresses within the same GPU, as well as introducing the packing/unpacking kernels again.

5 Conclusion and future work

On top of the end-user productivity and ease of use provided by the abstractions introduced, the framework provides numerous quantitative enhancements. Overall, this work has presented the following improvements:

1. Implicitly managing memory transfers and task execution dependencies across multiple hardware devices eliminates the need for code refactoring and decreases complexity.
2. Computing performance is boosted by up to 300% implicitly leveraging from hardware-specific optimizations on a single GPU.
3. A configurable scheduler that optimizes data locality and evenly distributes workload across multiple devices allows seamless utilization of multi-GPU nodes and attains linear scalability.
4. The potential heterogeneity awareness of the underlying network interface is automatically utilized, eliminating the need for explicitly written code to take advantage of this feature, providing up to three times better performance.
5. The portable abstractions for communication among distributed devices remove the need for explicit monitoring of device-host and inter-node transfers to ensure data consistency and completion. While this approach does incur some overheads within 10% of MPI for small messages, it also delivers exceptional performance for large messages, with gains of up to 20%.
6. Leveraging from these optimizations, a proxy 3-dimensional Jacobi application on top of PREMA achieved performance improvements of up to 30%.

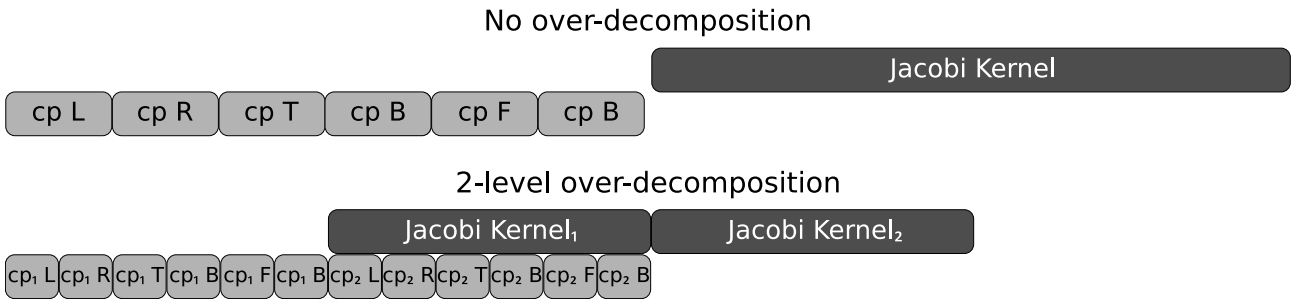

FIGURE 14
An example of the Jacobi3D execution timeline with no and 2-level over-decomposition. Each kernel invocation requires the copy of the six halo buffers, left (L), right (R), top (T), bottom (B), forward (F), and back (B), into the GPU. By decomposing the data domain (and the kernel invocation) into two pieces, PREMA can overlap the memory transfers of the second with the kernel invocation of the first, saving a significant amount of time. Note that the timeslots attributed to halos processing in the example are exaggerated to enhance clarity.

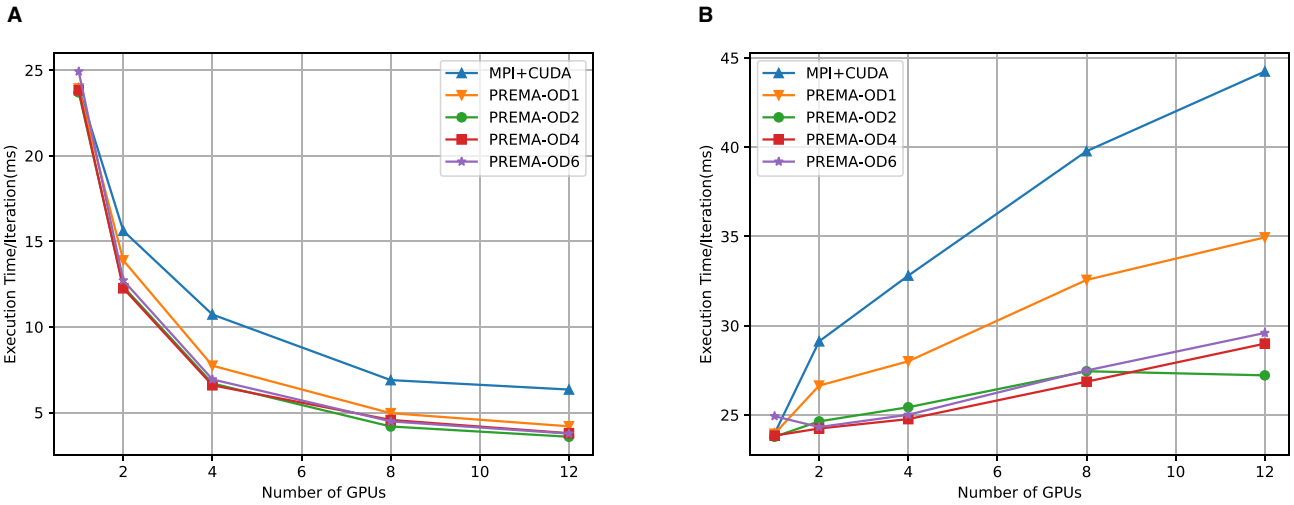

FIGURE 15
Strong **(A)** and weak **(B)** scaling performance of the Jacboid3D proxy application with different levels of over-decomposition.

7. Introducing over-decomposition further increases the distributed Jacobi's performance by 40% compared to the MPI+CUDA implementation by allowing implicit operation pipe-lining and latency overlapping.

In the future, we plan to improve the tasking framework's performance further, introduce more sophisticated schedulers, and provide compiler support to generate device kernels automatically. Moreover, we intend to extend PREMA's implicit load balancing layer to accommodate the workload of heterogeneous devices and experiment with a range of load balancing and scheduling policies.

Data availability statement

The raw data supporting the conclusions of this article will be made available by the authors, without undue reservation.

Author contributions

PT: Conceptualization, Data curation, Investigation, Methodology, Software, Writing – original draft, Writing – review & editing. NC: Conceptualization, Funding acquisition, Investigation, Methodology, Project administration, Resources, Supervision, Validation, Writing – review & editing.

Funding

The author(s) declare financial support was received for the research, authorship, and/or publication of this article. This work was partly funded by the Dominion Fellowship, the Richard T. Cheng Endowment at Old Dominion University, and the NSF grants: CCF-1439079 and CNS-1828593.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's note

References

Ang, J., Chien, A. A., Hammond, S. D., Hoisie, A., Karlin, I., Pakin, S., et al. (2021). *Reimagining codesign for advanced scientific computing: Report for the ASCR workshop on reimagining codesign*. Technical Report. USDOE Office of Science. doi: 10.2172/1843574

Ashbaugh, B., Bader, A., Brodman, J., Hammond, J., Kinsner, M., Pennycook, J., et al. (2020). "Data parallel c++: enhancing sycl through extensions for productivity and performance," in *Proceedings of the International Workshop on OpenCL, IWOCL '20, New York, NY, USA* (Association for Computing Machinery). doi: 10.1145/3388333.3388653

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P. A. (2011). StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput.* 23, 187–198. doi: 10.1002/cpe.1631

Baden, S. B., Gannon, D. B., Norman, M. L., and Chrisochoides, N. P. (1999). *Structured Adaptive Mesh Refinement (Samr) Grid Methods*. Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-1-4612-1252-2

Balasubramaniam, M., Barker, K., Banicescu, I., Chrisochoides, N., Pabico, J., and Carino, R. (2004). "A novel dynamic load balancing library for cluster computing," in *Proceedings 3rd International Symposium on Parallel and Distributed Computing*, 346–353.

Barker, K., Chernikov, A., Chrisochoides, N., and Pingali, K. (2004). A load balancing framework for adaptive and asynchronous applications. *IEEE Trans. Parallel Distr. Syst.* 15, 183–192. doi: 10.1109/TPDS.2004.1264800

Barker, K., Chrisochoides, N., Nave, D., Dobellaere, J., and Pingali, K. (2002). Data movement and control substrate for parallel adaptive applications. *Concurr. Comput.* 14, 77–105. doi: 10.1002/cpe.617

Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). "Legion: expressing locality and independence with logical regions," in *Proceedings International Conference on High Performance Computing, Networking, Storage and Analysis*, 1–11. doi: 10.1109/SC.2012.71

Beckingsale, D. A., Burmark, J., Hornung, R., Jones, H., Killian, W., Kunen, A. J., et al. (2019). "RAJA: portable performance for large-scale scientific applications," in *Proceedings IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, 71–81. doi: 10.1109/P3HPC49587.2019.00012

Bosilca, G., Bouteiller, A., Danalis, A., Herault, T., Lemarinier, P., and Dongarra, J. (2012). DAGuE: a generic distributed DAG engine for high performance computing. *Parallel Comput.* 38, 37–51. doi: 10.1016/j.parco.2011.10.003

Bozkus, Z., Choudhary, A., Fox, G., Haupt, T., and Ranka, S. (1993). "Fortran 90d/hpf compiler for distributed memory mimd computers: design, implementation, and performance results," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, Supercomputing '93* (New York, NY, USA: Association for Computing Machinery), 351–360. doi: 10.1145/169627.169750

Carter Edwards, H., Trott, C. R., and Sunderland, D. (2014). Kokkos: enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel Distr. Comput.* 74, 3202–3216. doi: 10.1016/j.jpdc.2014.07.003

Cavé, V., Zhao, J., Shirako, J., and Sarkar, V. (2011). "Habanero-Java: the new adventures of old X10," in *Proceedings 9th International Conference on Principles and Practice of Programming in Java*, 51–61. doi: 10.1145/2093157.2093165

Chamberlain, B., Callahan, D., and Zima, H. (2007). Parallel programmability and the Chapel language. *Int. J. High Perform. Comput. Appl.* 21, 291–312. doi: 10.1177/1094342007078442

Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., et al. (2005). X10: an object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.* 40, 519–538. doi: 10.1145/1103845.1094852

Chernikov, A. N., and Chrisochoides, N. P. (2008). "Three-dimensional delaunay refinement for multi-core processors," in *Proceedings International Conference on Supercomputing*, 214–224. doi: 10.1145/1375527.1375560

Chrisochoides, N. (1995). *PREMA: Portable runtime environment for multicomputer architectures*. Available online at: https://web.archive.org/web/19970609213320/cs.cornell.edu/Info/People/nikosc/projects/prema/index.html (accessed March, 2024).

Chrisochoides, N. (1996). Multithreaded model for the dynamic load-balancing of parallel adaptive PDE computations. *Appl. Numer. Mathem.* 20, 349–365. doi: 10.1016/0168-9274(95)00104-2

Chrisochoides, N. (1998). "Parallel run-time system for adaptive mesh refinement," in *Proceedings Solving Irregularly Structured Problems in Parallel*, 396–405. doi: 10.1007/BFb0018556

Chrisochoides, N. (2005). Parallel mesh generation. *Numer. Solut. Part. Differ. Equat. Parallel Comput.* 51, 237–259. doi: 10.1007/3-540-31619-1_7

Chrisochoides, N., Barker, K., Nave, D., and Hawblitzel, C. (2000). Mobile object layer: a runtime substrate for parallel adaptive and irregular computations. *Adv. Eng. Softw.* 31, 621–637. doi: 10.1016/S0965-9978(00)00032-6

Chrisochoides, N., Fox, G., and Haupt, T. (1994). "A computational toolkit for colliding black holes and CFD," in *Fluid Dynamics Conference*. doi: 10.2514/6.1994-2249

Chrisochoides, N., and Hawblitzel, C. (1998). "Data migration substrate for the load balancing of parallel adaptive unstructured mesh computations," in *Proceedings 6th Int'l Conf. on Numerical Grid Generation in Computational Field Simulation*.

Chrisochoides, N., Kodukula, I., and Pingali, K. (1997). "Data movement and control substrate for parallel scientific computing," in *Proceedings Communication and Architectural Support for Network-Based Parallel Computing*, 256–268. doi: 10.1007/3-540-62573-9_19

Chrisochoides, N. P. (2016). "Telescopic approach for extreme-scale parallel mesh generation for CFD applications," in *Proceedings 46th AIAA Fluid Dynamics Conference*, 3181. doi: 10.2514/6.2016-3181

Drakopoulos, F., Tsolakis, C., and Chrisochoides, N. P. (2019). Fine-grained speculative topological transformation scheme for local reconnection methods. *AIAA J.* 57, 4007–4018. doi: 10.2514/1.J057657

Duran, A., Ayguadé, E., Badia, R. M., Labarta, J., Martinell, L., Martorell, X., et al. (2011). Ompss: a proposal for programming heterogeneous multi-core architectures. *Parallel Proc. Lett.* 21, 173–193. doi: 10.1142/S0129626411000151

Fedorov, A., and Chrisochoides, N. (2004). "Location management in object-based distributed computing," in *Proceedings IEEE International Conference on Cluster Computing*, 299–308.

Foteinos, P. A., and Chrisochoides, N. P. (2014). High quality real-time image-to-mesh conversion for finite element simulations. *J. Parallel Distr. Comput.* 74, 2123–2140. doi: 10.1016/j.jpdc.2013.11.002

Fox, G., Ranka, S., Scott, M., Malony, A., Browne, J., Chen, M., et al. (1993). "Common runtime support for high-performance parallel languages parallel compiler runtime consortium," in *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 752–757.

Garner, K., Thomadakis, P., Kennedy, T., Tsolakis, C., and Chrisochoides, N. (2019). "On the end-user productivity of a pseudo-constrained parallel data refinement method for the advancing front local reconnection mesh generation software," in *Proceedings AIAA Aviation Forum 2019*. doi: 10.2514/6.2019-2844

Garner, K., Tsolakis, C., Thomadakis, P., and Chrisochoides, N. (2024). "Towards distributed speculative adaptive anisotropic parallel mesh generation," in *AIAA Aviation Forum 2024*.

Huang, T.-W., Lin, D.-L., Lin, C.-X., and Lin, Y. (2022). Taskflow: a lightweight parallel and heterogeneous task graph computing system. *IEEE Trans. Parallel Distrib. Syst.* 33, 1303–1320. doi: 10.1109/TPDS.2021.3104255

Kaiser, H., Heller, T., Adelstein-Lelbach, B., Serio, A., and Fey, D. (2014). "HPX: a task based programming model in a global address space," in *Proceedings 8th International Conference on Partitioned Global Address Space Programming Models*, 1–11. doi: 10.1145/2676870.2676883

Kale, L. V., and Krishnan, S. (1993). Charm++: a portable concurrent object oriented system based on C++. *SIGPLAN Not.* 28, 91–108. doi: 10.1145/167962.165874

Kot, A., Chernikov, A., and Chrisochoides, N. (2011). "The evaluation of an effective out-of-core run-time system in the context of parallel mesh generation," in *IEEE International Parallel and Distributed Processing Symposium*, 164–175. doi: 10.1109/IPDPS.2011.25

Laboratory-UIUC, P. P. (2022). *Charm++ jacobi3D proxy*. Available online at: https://github.com/UIUC-PPL/charm/tree/jchoi/hips22/examples/charm++/cuda/gpudirect/jacobi3d/mpi (accessed March, 2024).

Majeti, D., and Sarkar, V. (2015). "Heterogeneous Habanero-C (h2c): a portable programming model for heterogeneous processors," in *Proceedings IEEE*

International Parallel and Distributed Processing Symposium Workshop, 708–717. doi: 10.1109/IPDPSW.2015.81

Nave, D., Chrisochoides, N., and Chew, L. (2004). Guaranteed-quality parallel delaunay refinement for restricted polyhedral domains. *Comput. Geomet.* 28, 191–215. doi: 10.1016/j.comgeo.2004.03.009

OpenMPI (2024). *Faq: Running cuda-aware open MPI*. Available online at: https://www.open-mpi.org/faq/?category=runcuda (accessed May, 2024).

Parashar, M., Hariri, S., Haupt, T., and Fox, G. C. (1994). *Design of an application development toolkit for hpf/fortran 90d*. Northeast Parallel Architecture Center. 93. Available online at: https://surface.syr.edu/npac/93 (accessed March, 2024).

Seo, S., Amer, A., Balaji, P., Bordage, C., Bosilca, G., Brooks, A., et al. (2016). "Argobots: a lightweight threading/tasking framework," in *IEEE Transactions on Parallel and Distributed Systems*.

Thomadakis, P., and Chrisochoides, N. (2023). Toward runtime support for unstructured and dynamic exascale-era applications. *J. Supercomput.* 38, 4675-4695. doi: 10.1007/s11227-022-05023-z

Thomadakis, P., Tsolakis, C., and Chrisochoides, N. (2022). Multithreaded runtime framework for parallel and adaptive applications. *Eng. Comput.* 38, 4675–4695. doi: 10.1007/s00366-022-01713-7

Thomadakis, P., Tsolakis, C., Vogiatzis, K., Kot, A., and Chrisochoides, N. (2018). "Parallel software framework for large-scale parallel mesh generation and adaptation for cfd solvers," in *AIAA Aviation Forum 2018* (Atlanta, Georgia). doi: 10.2514/6.2018-2888

Tsolakis, C., Thomadakis, P., and Chrisochoides, N. (2022). Tasking framework for adaptive speculative parallel mesh generation. *J. Supercomput.* 78, 1–32. doi: 10.1007/s11227-021-04158-9

von Eicken, T., Culler, D. E., Goldstein, S. C., and Schauser, K. E. (1992). Active messages: a mechanism for integrated communication and computation. *SIGARCH Comput. Archit. News* 20, 256–266. doi: 10.1145/146628.140382