A Pluggable Framework for Composable HPC Scheduling Libraries

Max Grossman*, Vivek Kumar†, Nick Vrvilo*, Zoran Budimlić*, and Vivek Sarkar*
*Rice University, {max.grossman, nick.vrvilo, zoran, vsarkar}@rice.edu
†Indraprastha Institute of Information Technology Delhi, vivekk@iiitd.ac.in

***Abstract*—Driven by the increasing diversity of current and future HPC hardware and software platforms, the HPC community has seen a dramatic increase in research and development efforts into the composability of discrete software systems. While modularity is often desirable from a software engineering, quality assurance, and maintainability perspective, the barriers between software components often hide optimization opportunities. Recent examples of work in composable HPC software include GPU-Aware MPI, OpenMP's target directive, Lithe, HCMPI, and MVAPICH's unified communication runtime. These projects all deal with breaking down the walls between software or hardware components in order to achieve performance, programmability, and/or portability gains. However, they also generally focus on composing only specific types of HPC software and have limited extensability.**

In this paper, we present work on using a pluggable API framework on top of a "generalized work-stealing" runtime to achieve composability of communication, accelerator, and other HPC libraries. We motivate this work by the increasing heterogeneity of HPC hardware, software, and applications, and note that as heterogeneity increases many discrete software frameworks will need to cooperate within a single process. Our framework, called HiPER (a Highly Pluggable, Extensible, and Re-configurable scheduling framework for HPC) enables exactly this cooperation.

We demonstrate the programmability improvements enabled by the HiPER framework through the use of novel APIs which reduce programmer burden. We also present performance studies that demonstrate that through unified and asynchronous scheduling of composed software systems we can achieve performance improvements over hand-optimized benchmarks.

***Keywords*-composability; HPC; libraries; accelerators; GPU; CUDA; OpenSHMEM; MPI; UPC++; unified; heterogeneous;**

I. Introduction & Motivation

The number and diversity of software libraries and hardware components used by scientific applications on high-performance hardware has steadily increased in the last decade. At the same time, the scale of high-performance computing (HPC) platforms has steadily increased both in terms of the number of processing cores and shared-memory nodes, making it more important for programming systems to hide latencies using asynchronous APIs.

However, the use of multiple software modules and the ability to hide inter- or intra-node latencies are directly in conflict. Software modules are by definition discrete and separate, with no knowledge of the other software tenants in a multi-tenant software system. As a result, they generally offer little or no support for expressing the connection between operations in two separate modules. This generally leads to the need for overly coarse synchronization when algorithmic dependencies cross module boundaries, inhibiting latency-hiding techniques that generally rely on exposing the maximal amount of parallelism in an application.

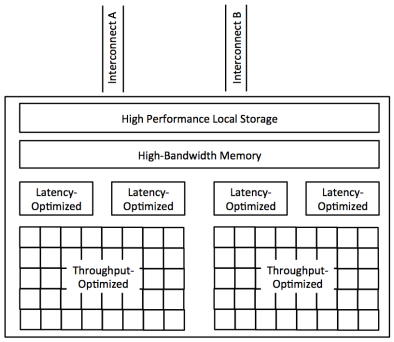

Figure 1. Depiction of the abstract platform motivating this work.

In this paper, we explore the design and implementation of a highly extensible HPC runtime called HiPER (Highly Pluggable, Extensible, and Re-configurable scheduling framework for HPC). HiPER unifies the representation of computation, communication, and other work as tasks in a task-parallel runtime system. It enables the unified scheduling of a full application workload on a single runtime, supports third-party extensions to the type of work it schedules, and emphasizes future-based APIs for maximal asynchrony.

A. Our Abstract Platform Model of Future HPC Systems

To motivate the design presented in Section II we briefly present an abstract platform model representative of future HPC systems. Figure 1 summarizes this model.

The computational backbone of future HPC systems will be a pool of bandwidth-optimized, lightweight, programmable cores [1]. This pool of lightweight cores in each node will be paired with zero or more latency-optimized cores. While these latency-optimized cores may be used for some computational kernels, their primary use will be the orchestration of work on the lightweight cores, communication over the network, I/O to local storage (e.g. for checkpointing), and other management functions.

The management cores and computational cores will most likely share and have direct access to high-bandwidth, high-latency memory. While this high-bandwidth memory

may be accessed through load and store instructions, the use of specialized data transfer APIs and on-chip, lower-latency memory will enable locality optimizations for both management and computational cores.

A high-performance, non-uniform interconnect will connect shared-memory nodes, similar to existing systems. This interconnect may be accessed through multiple software libraries in a single application (e.g. MPI, UPC++ [2], OpenSHMEM [3]) depending on how library abstractions fit application communication patterns.

Each shared-memory node will include at a minimum some high-bandwidth (likely Flash-based) local storage and also have access to a higher-latency, lower-bandwidth shared filesystem. In addition, future systems may include non-volatile memory (NVM) for use as a persistent storage for checkpoint-restart, as a Burst Buffer [4], or in other configurations. Again, a custom library will be required for direct access to NVM.

While the number of total cores in these future systems will almost certainly increase, the number of heavyweight, management cores may actually decrease relative to today's homogeneous x86-based systems. For example, the Sunway TaihuLight [5] has only eight management cores per node. Hence, in the future it will be even more crucial to efficiently utilize these management cores using techniques like the ones presented in this paper.

B. Contributions

Motivated by the abstract platform model described in Section I-A, the diversification in software systems that will be necessary to program future implementations of this platform model, and the conflict between software modularization and cross-module optimization, we introduce the HiPER system in this paper and make the following contributions:

1) A "generalized work-stealing" scheduler and its use in unified scheduling of heterogeneous workloads on heterogeneous systems.
2) A pluggable and extensible task-based programming model which enables the composition of multiple computational and communication libraries on top of a generalized work-stealing runtime.
3) A description of an implementation of the above two concepts on existing HPC systems, including the implementation of pluggable modules for the CUDA, MPI, OpenSHMEM, and UPC++ HPC libraries.
4) An evaluation of the performance and programmability of the ISx [6], HPGMG [7], UTS [8], Graph500 [9], and a geophysical application using our HiPER implementation.

The source code for HiPER and all benchmarks used in this paper are also made available open source at [10].

The remainder of this paper is structured as follows. Section II presents the design and implementation of the HiPER system. Section III evaluates the programmability and performance of our implementation across a range of HPC systems. Section IV summarizes related research. Finally, Section V discusses how HiPER might evolve in the future in response to changes in the HPC landscape and Section VI presents our conclusions from this work.

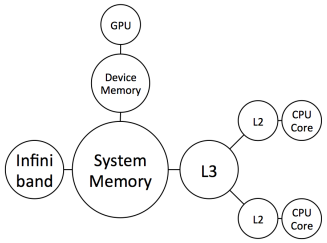

Figure 2. An example of the HiPER Platform Model.

II. Design & Implementation

At a high level, the HiPER system consists of three components: 1) a platform model, 2) a generalized work-stealing, multi-threaded runtime, and 3) pluggable, third-party software modules.

The HiPER Platform Model offers an abstraction of the heterogeneous hardware resources across which the workload of an application will be distributed. The Generalized Work-Stealing Runtime manages load-balancing and execution of user-created tasks placed at different locations in the Platform Model. The Pluggable Software Modules sit on top of the runtime and expose familiar APIs to the user (e.g. MPI, OpenSHMEM) while placing tasks in the HiPER Platform Model to be executed by the work-stealing runtime.

A. HiPER Platform Model

The HiPER Platform Model consists of an undirected, unweighted graph. Nodes within the graph logically represent hardware components that software libraries may utilize, and are referred to as "places" [11]. Figure 2 depicts an example HiPER Platform Model.

Edges between places in the platform graph logically represent direct accessibility between hardware components. For example, a direct edge between system memory and GPU device memory indicates that data in system memory is directly transferrable to that GPU's device memory. There is no strict requirement that there be a one-to-one mapping of places or edges in the platform model to physical hardware or connections. However, some similarities are likely desirable for performance fidelity.

The HiPER Platform Model is implemented as an in-memory graph structure. It is loaded from a JSON-formatted file at HiPER runtime initialization. HiPER comes with utilities for automatically generating JSON platform configuration files using the HWloc library [12], but users are also free to edit these configurations.

B. Generalized Work-Stealing Runtime

Work-stealing is a common technique for automatic load balancing across homogeneous cores [13]. At a high level, work-stealing balances work across a persistent thread pool by having idle threads "steal" tasks from work pools belonging to neighboring threads.

The "Generalized" in "Generalized Work-Stealing" refers to the ability to perform work-stealing load balancing for more than homogeneous computational tasks. The HiPER Generalized Work-Stealing runtime depends upon the Platform Model and consists of four components: a set of persistent worker threads, task deques of eligible tasks at each place in the platform model graph, a pop and steal path for each thread which traverses some subset of the places in the platform model, and an API for enqueueing tasks to the task deques in the platform model. This work does not make any novel contributions in the area of load-balancing policies.

1) Persistent Thread Pool: Like most work-stealing runtimes, our generalized work-stealing runtime sits on persistent set of worker threads on which all tasks are executed. These worker threads reside on the management cores of an HPC system. The number of worker threads to create is defined in the JSON file used to initialize the platform model, and generally equals the number of management cores.

Tasks are defined as suspendable single-threaded streams of execution, and may synchronize on other tasks or create new tasks.

As in Realm [14], HPX [15], and QThreads [16], the HiPER runtime threads use runtime-managed call stacks to enable task suspension. When a HiPER task blocks on a synchronization operation, HiPER will suspend that task without blocking a CPU core on it by swapping its call stack off of the current thread, wrapping its continuation in a task, and making the execution of that task predicated on the satisfaction of the appropriate synchronization event. Call stack suspension relies on the Boost.Context library [17].

2) Per-Place Task Deques: Each place in the platform model includes N task deques, where N is the number of threads in the persistent thread pool described above. The ith deque in a place contains only eligible tasks that are ready to begin executing and which were spawned by the ith worker thread. Hence, given a place and a thread looking for work to do it is straightforward to differentiate between tasks created by that same thread and tasks created by other threads. Executing a task created by the same thread likely encourages locality, while executing a task created by other threads encourages load balancing.

3) Per-Thread Pop and Steal Paths: Each worker thread has one "pop path" and one "steal path". Each of these paths is an ordered list of places in the platform model. A path defines the sequence of places a runtime thread will traverse when searching for a task to execute. When traversing a pop path, a runtime thread will only check for work that it created.

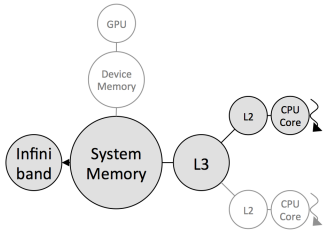

Figure 3. An example of a pop or steal path through the HiPER Platform Model.

A steal path is similar, but runtime threads traversing a steal path will only look for work created by other runtime threads. Figure 3 depicts an example path through the platform model from Figure 2. Pop and steal paths are also loaded from the platform configuration JSON file.

Hence, each runtime thread's logic simply consists of:

1) Search along its pop path for any work created by the same thread at any place.
2) If no work has been found yet, search along its steal path, only looking for work created by other threads.
3) Repeat #2 until either work is found or a runtime shutdown signal is received by this thread.

When a runtime thread discovers a task along either its pop or steal path, that task is immediately executed.

These paths are infinitely flexible, and so can be used to encode any number of load balancing policies. For example, a memory hierarchy-aware policy could be set by having runtime threads traverse up the logical memory hierarchy represented by the platform model. However, this is not a property of pop and steal paths themselves but rather one possible result of their flexibility.

4) Task Creation APIs: The generalized work-stealing runtime must also expose APIs for placing and removing tasks in deques in the platform model. These APIs may be used by a programmer, but are also key to implementing the pluggable HiPER modules described in Section II-C. HiPER currently only supports C++ APIs.

The `async` API creates a task executing `body` at the place closest to the current runtime thread:

```
async([] { body; });
```

The `async_at` API creates a task executing `body` at a specific place:

```
async_at([] { body; }, place);
```

HiPER's API and runtime also support the use of promises and futures for inter-task synchronization. A promise in HiPER is a single-assignment, thread-safe container for some value. A future is a read-only handle on that value. Promises and futures can serve as a flexible, point-to-point synchronization channel from one source task to many sink tasks. Sink tasks may block on the future, only being released when another task performs a put on the associated promise.

Promise and future objects can be created in HiPER using standard C++ constructors and getters:

```
promise_t *p = new promise_t();
future_t f = p->get_future();
```

Satisfying a promise, blocking on a future, and fetching the put value from a future are simple member function calls:

```
p->put(nullptr);
f->wait();
val = f->get();
```

Additionally, the `async_future` API creates a task and returns a future which will be automatically satisfied when that task completes, while the `async_await` API creates a task whose execution is predicated on the satisfaction of a given future object:

```
future = async_future([] { body; });
async_await([] { body; }, future);
```

Bulk task synchronization is possible using the `finish` API. `finish` waits for all tasks created in `body` before returning, including transitively spawned tasks.

```
finish([] { body; });
```

Many combined variants of the task creation APIs exist as well. For example, `async_future_await` creates a task whose execution is predicated on the satisfaction of a future, and returns a future that is satisfied when that task completes.

HiPER also comes with an `async_copy` API which asynchronously transfers data from a memory location in one place to a memory location in another place:

```
async_copy(dst_loc, dst_place, src_loc,
    src_place, nbytes);
```

C. Pluggable Software Modules

The final component of the HiPER system is its pluggable modules. A single pluggable module adds user-visible APIs that can be called to schedule module-specific tasks on the HiPER work-stealing runtime. These tasks may perform arbitrary logic. For example, an MPI module would extend the HiPER user-visible APIs with functions from the MPI standard. This would enable both 1) composability of familiar MPI APIs with other HiPER modules, and 2) unified scheduling of MPI communication with other work on the HiPER runtime. A complete HiPER module includes:

1) A module initialization function registered with the HiPER runtime which is called once during the life of a process.
2) A module finalization function registered with the HiPER runtime which is called once during the life of a process.
3) A set of optional, special-purpose functions registered with the HiPER runtime. For example, a module may register itself as responsible for handling data transfers between places of certain types in the platform model.
4) A set of functions added to the global HiPER namespace and accessible to programmers. These functions extend the capabilities of HiPER to make use of a new hardware or software component (e.g. GPUs, MPI, hard disks). These user-facing functions are commonly implemented by placing special-purpose, asynchronous tasks at special-purpose nodes in the platform model. As a result, all work created by HiPER modules is scheduled together on a single unified runtime. Examples of modules supported today include modules for CUDA, MPI, OpenSHMEM, and UPC++.

One of the key characteristics of HiPER modules is that they do not require that the software or hardware component they support be aware of HiPER or of the other HiPER modules.

A HiPER module is not part of the core HiPER runtime. It therefore can be implemented by any third-party HiPER user. Implementers of HiPER module are free to keep or change the semantics of the HPC libraries they are adding to HiPER's namespace. For example, there is no way for HiPER to enforce that a HiPER MPI module does not change the semantics of an MPI API it exposes. However, in general we suggest that modules maintain similar or identical semantics for standard APIs and create new APIs for implementing novel functionality.

To illustrate these points, we will perform several case studies on existing HiPER modules below. Note that in general, these HiPER modules may only implement a useful subset of the APIs they are implementing (e.g. the MPI module implements a subset of the MPI standard) and are not necessarily full, specifications-compliant implementations of the corresponding HPC libraries.

1) MPI Module: The MPI module implements a subset of the APIs in the MPI standard, relying on a full MPI library to handle the actual messaging (e.g. OpenMPI, MVAPICH, etc.). For this communication module and the others described below, no semantic changes are made to ordering or collective requirements. For example, for all collectives a single task from each MPI rank is expected to participate. Within a rank, if an `MPI_Send` in one task must be issued before an `MPI_Recv` in another task to prevent deadlock, it is the programmer's responsibility to ensure the correct ordering of those tasks.

Regarding the platform model and thread configuration, the MPI module relies on a single "Interconnect" place existing in the platform model and that place being on a single thread's pop and steal paths. This allows the MPI module to configure the underlying MPI implementation in `MPI_THREAD_FUNNELED` mode, keeping MPI runtime overheads low. It is up to individual modules to make these assertions about the current platform model during module initialization.

Many MPI APIs are implemented using the following flow, which we refer to as "taskifying":

1) A C++ lambda is created which captures the inputs to the MPI API being implemented, and which calls the underlying MPI library's implementation of that API.
2) This lambda is passed to the HiPER `async_at` API described in Section II-B4, targeting the Interconnect place in the platform place graph.
3) A `finish` scope is used to block the calling task on the completion of the spawned task. Under the covers, this deschedules the calling task until the spawned MPI task completes (i.e. creates a continuation).
4) At some time in the future, a runtime thread with the Interconnect place on its pop or steal path discovers the task created by step (2) and executes it. The continuation is then eligible for execution. Note that this is not a dedicated communication thread and may search other places before finding work at the Interconnect place.

For example, the HiPER Module implementation of `MPI_Send` is shown below:

```
finish([&] {
    async_at([&] {
        ::MPI_Send(buf, count, datatype,
            dest_rank, tag, comm);
    }, interconnect);
});
```

Asynchronous MPI APIs require a different approach. At the API level, the HiPER MPI module makes a small change to APIs like `MPI_Isend`, `MPI_Irecv`, etc. by removing the output `MPI_Request` argument which is normally used to query on the status of an asynchronous message, and replacing it by returning a `future_t` object whose satisfaction is predicated on the completion of the asynchronous MPI message. This `future_t` behaves as normal, supporting blocking `get` calls and tasks being predicated on its satisfaction via `async_await`.

In the runtime, asynchronous MPI APIs are implemented as follows:

1) The asynchronous MPI API is called directly, producing an `MPI_Request` object.
2) A new `promise_t` object is created, and it is stored with the `MPI_Request` object in a list of pending MPI operations.
3) A periodically polling asynchronous task is spawned which iterates over the list of pending MPI operations, finds any that have completed, and satisfies their associated `promise_t` objects. If after iterating through the pending list this polling task finds that there are still pending MPI operations, it yields to allow other useful work to be done before polling again. A polling task is not created if one already exists.
4) The `future_t` associated with the newly created `promise_t` is then returned from the HiPER MPI function. It can be used to register other HiPER work on the completion of the asynchronous MPI communication.

Using future-producing APIs like these enables programmers to compose MPI messages with other work in the system, such as task or accelerator parallelism. For example, the code snippet below would trigger a task on the receipt of an asynchronous MPI message.

```
fut = MPI_Irecv(...);
async_await([=] { body; }, fut);
```

Note that we make no assumptions about how asynchronous progression of communication is implemented in the various communication runtimes. However, the periodic polling on `MPI_Request` may give the MPI runtime opportunities to make forward progress.

2) OpenSHMEM Module: The current version of the OpenSHMEM specification (v1.3) makes no guarantees about thread safety. The development of a HiPER OpenSHMEM module enables the safe and standard-compliant use of OpenSHMEM in multi-threaded applications. Like MPI, the OpenSHMEM specification consists entirely of functions and is not object-oriented like UPC++. As a result, many of the supported OpenSHMEM APIs are implemented using the "taskify" pattern described along with the MPI module in Section II-C1.

The integration of OpenSHMEM into HiPER enabled the development of novel OpenSHMEM APIs. For example, the OpenSHMEM specification includes `wait` APIs which allow an OpenSHMEM process to block waiting on a remote put into its address space. While these are useful APIs for point-to-point synchronization, their blocking nature wastes CPU cycles and lowers application scalability. One extension to the OpenSHMEM APIs enabled as part of the HiPER module implementation was an asynchronous variant which makes a task's execution predicated on a put by a remote process, called `shmem_async_when`:

```
shmem_async_when(mem_addr, wait_for_val, [=] {
    body;
});
```

3) CUDA Module: The CUDA Module supports basic CUDA operations, such as blocking data transfers, asynchronous data transfers, and asynchronous CUDA kernels.

The CUDA Module is the only module discussed here which registers special-purpose functions with the HiPER runtime. In particular, it registers itself as handling copies to or from GPU places. Anytime a call to HiPER's `async_copy` API reads or writes a GPU place, it is automatically handed off to the CUDA Module.

The CUDA Module uses the same polling technique as the MPI Module (described in Section II-C1) to support asynchronous CUDA operations satisfying HiPER promises.

D. Example HiPER Usage

Consider a three-dimensional stencil application, in which the cells of a three-dimensional, regular grid are distributed

in only the z-direction among MPI ranks. Let us assume that in this simplified application, a single data-parallel kernel is run across the z values a given rank is responsible for before a halo exchange occurs with neighboring ranks. This process repeats on each of several time iterations.

In an MPI+OpenMP implementation, this application could be implemented as something like the following:

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in parallel
#pragma omp parallel for
  for (...) { }

  // Transmit ghost regions to neighbors,
  // and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
#pragma omp parallel for
  for (...) { }

  // Wait for all sends/recvs to complete
  MPI_Waitall(4, reqs);
}
```

Using MPI+CUDA instead produces a slightly longer code snippet. More importantly, doing so introduces more blocking operations which may waste host CPU cycles. Additionally, the inter-statement dependencies in the straight-line sequence of API calls is unclear as a result of a lack of composability between the CUDA and MPI APIs:

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in CUDA
  stencil<<<...>>>(...);

  // Copy ghost region from CUDA device
  cudaMemcpy(..., cudaMemcpyDeviceToHost);

  // Transmit ghost regions to neighbors,
  // and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
  stencil<<<...>>>(...);

  // Wait for all transmissions to complete
  MPI_Waitall(4, reqs);

  // Copy received ghost region to CUDA device
  cudaMemcpy(..., cudaMemcpyHostToDevice);
}
```

However, it may also be possible to improve performance by combining MPI, OpenMP, and CUDA by processing the smaller ghost region with OpenMP to avoid a `cudaMemcpy` while still offloading the main computational region to CUDA. The code snippet below not only requires that the programmer have expertise in OpenMP, CUDA, and MPI, but also understand how to manage their interaction safely.

```
for (t = 0; t < nt; t++) {
  // Process ghost regions on this rank in parallel
#pragma omp parallel for
  for (...) { }

  // Transmit ghost regions to neighbors,
  // and post receives
  MPI_Isend(..., &reqs[0]);
  MPI_Isend(..., &reqs[1]);
  MPI_Irecv(..., &reqs[2]);
  MPI_Irecv(..., &reqs[3]);

  // Process remainder of z values on this rank
  stencil<<<...>>>(...);

  // Wait for all transmissions to complete
  MPI_Waitall(4, reqs);

  // Copy received ghost region to CUDA device
  cudaMemcpy(..., cudaMemcpyHostToDevice);
}
```

In contrast, expressing the same computational pattern in HiPER's future-based, composable programming model would look like the following (assuming the user already has the CUDA and MPI modules installed):

```
for (t = 0; t < nt; t++) {
  // Place an outer finish scope to ensure all
  // work completes before continuing to the next
  // time step
  finish([&] {

    // Asynchronously process ghost regions on
    // this rank in parallel
    ghost_fut = forasync_future([] (z) { ... });

    // Asynchronously exchange ghost regions with
    // neighbors
    reqs[0] = MPI_Isend_await(..., ghost_fut);
    reqs[1] = MPI_Isend_await(..., ghost_fut);
    reqs[2] = MPI_Irecv(...);
    reqs[3] = MPI_Irecv(...);

    // Asynchronously process remainder of z
    // values on this rank
    forasync_cuda(..., [] (z) { ... });

    // Copy received ghost region to CUDA device
    async_copy_await(..., reqs[2], reqs[3]);
  });
}
```

Note that in the code listing above, dependencies are expressed more naturally and between different software components. Each asynchronous operation waits on precisely the futures it needs to in order to ensure its dependencies are maintained, input/output relations are visible as return values and API parameters, and blocking operations do not actually block CPU threads. At the same time, the future-based APIs

used to express CUDA parallelism and MPI communication remain syntactically similar to their standard variants in order to take advantage of existing programmer expertise.

III. Experimental Evaluation

A. Experimental Setup

The experiments in this section were run on one of two platforms: the Edison supercomputer at NERSC or the Titan supercomputer at ORNL. Edison is a Cray XC30 with 2×12-core Intel Ivy Bridge CPUs and 64 GB DDR3 in each node. Titan is a Cray XK7 with a 16-core AMD CPU, an NVIDIA K20X, and 32GB of DRAM in each node. For the experiments listed below, Cray SHMEM v7.4.0 and GCC v4.9.3 were used on Titan. GCC 5.2.0 was used on Edison. All flat UPC++, MPI, or OpenSHMEM experiments are run with 1 process pinned to each core. All hybrid experiments on Edison are run with 2 processes and 12 threads per process, and on Titan are run with 1 process and 16 threads per process. All tests are repeated ten times, and error bars shown represent 95% confidence intervals.

Our benchmark suite consists of:

1) HPGMG-FV [7]: "Implements full multigrid algorithms using finite-volume... methods". Uses the UPC++ and MPI modules. This is a weak scaling benchmark, and was run with `log2_box_dim=7` and `target_boxes_per_rank=8` based on the advice of the HPGMG-FV developers.
2) ISx [6]: Integer sort benchmark. Uses the OpenSHMEM module. This is a weak scaling benchmark, and was run with 2^{29} keys to sort per process.
3) GEO: A three-dimensional stencil application for geophysical subsurface imaging. Uses the CUDA and MPI modules, and tests weak scaling.
4) UTS [8]: Unbalanced tree search. Uses the OpenSHMEM module. This is a strong scaling benchmark, and it was run with the T1XXL dataset.
5) Graph500 [9]: Parallel, distributed breadth first search of a graph. Uses the MPI module. This is a strong scaling benchmark, and was run using 2^{31} vertices with edge factor set to 16.

Space limitations prevent us from going into detail on the implementation of each benchmark. However, note that all benchmark code is available at [10].

B. Regular Workloads

Figures 4, 5, and 6 depict the weak-scaling of HPGMG-FV, ISx, and GEO on Titan. We note that for HPGMG-FV and ISx the HiPER and reference hybrid implementations are comparable in performance. For GEO, HiPER consistently improves performance by ~2% on average by reducing blocking CUDA operations through future-based programming. The Flat OpenSHMEM implementation of ISx outperforms the two hybrid versions at smaller node counts, it scales poorly to 512 and 1024 nodes due to a global all-to-all.

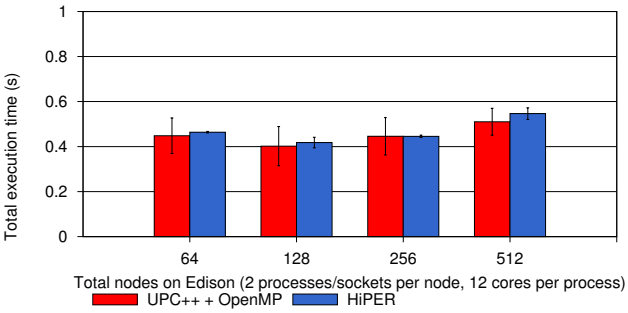

Figure 4. Total HPGMG solve time on up to 512 Edison nodes.

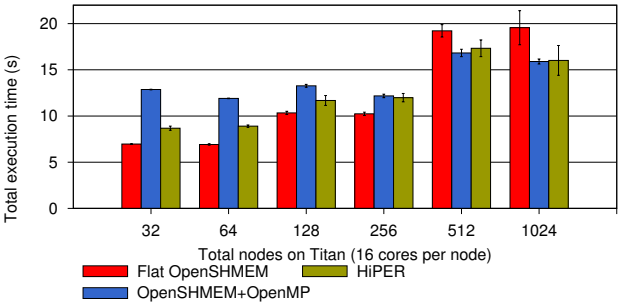

Figure 5. Total ISx execution time. Weak scaling up to 1024 nodes on Titan.

C. Irregular Workloads

We now turn to more irregular benchmarks, UTS and Graph500. Due to the asynchrony of HiPER's APIs and the ability to more naturally express the algorithmic dependencies of a parallel algorithm, we expect HiPER to perform better on these types of workloads.

1) UTS: Figure 7 shows the overall execution time of UTS using OpenSHMEM+OpenMP, OpenSHMEM+OpenMP Tasks, and AsyncSHMEM. The OpenSHMEM+OpenMP Tasks and AsyncSHMEM versions of this benchmark are identical in the structure of their parallelism. All three versions use manual, application-level, distributed load bal-

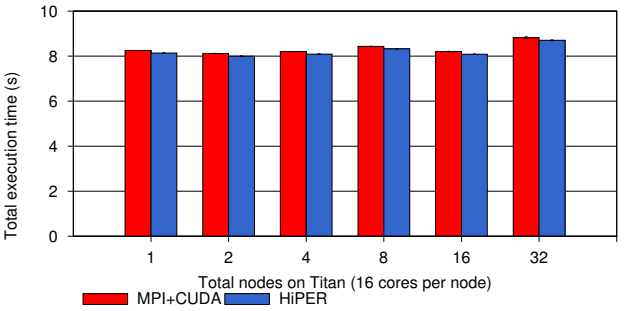

Figure 6. Total GEO execution time. Weak scaling up to 32 nodes on Titan.

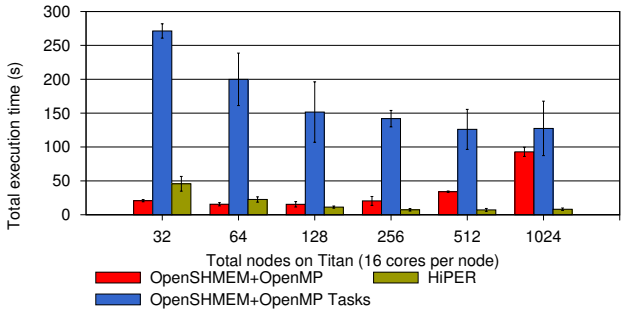

Figure 7. Total UTS execution time.

ancing, as do the reference UTS implementations.

The hand-coded OpenSHMEM+OpenMP version of UTS scales similarly to HiPER up to 128 nodes, but starts to degrade as contention from distributed load balancing increases.

Because of the lack of integration between OpenSHMEM and OpenMP, the OpenSHMEM+OpenMP Tasks version must repeatedly use coarse-grain synchronization to wait on all pending tasks before checking for completion and performing distributed load balancing.

2) Graph500: We implement a version of Graph500 in HiPER based on the work in [18]. While we observe little performance improvement to-date, the programmability benefits have been significant. Both the reference Graph 500 implementations and the work in [18] must constantly poll for incoming data from remote processes. This polling adds overhead, and significantly complicates the implementation.

In our implementation of Graph500, we are able to use the novel API introduced with this work, `shmem_async_when`, to offload that polling to the HiPER runtime. Further details are provided in [19].

IV. Related Work

Past work has already explored the use of thread offload for inter-node communication using techniques related to this work [20–23]. These works demonstrated the performance and scalability benefits of a dedicated communication thread to which all communication is funneled; however, they were all hard-coded to a single communication library (MPI or UPC++), dedicated an entire OS thread to communication (hurting the performance of more computationally-bound applications), and in the case of [23] depended on changes to the MPI runtime itself.

GPU-Aware MPI [24] enabled the direct communication of data from a GPU sitting in one node of a cluster to another GPU sitting in a different node using MPI APIs. This work demonstrated both performance benefits from a more direct data path and programmability benefits from using a single asynchronous MPI call. While this work is restricted to composing NVIDIA GPUs and MPI, the techniques used are generally applicable. A future direction of the HiPER project would allow registered modules to query for other modules which they can integrate with.

Similarly, MPI-ACC [25, 26] enables direct, inter-node, inter-GPU communication at the API level. Unlike GPU-Aware MPI, the underlying MPI-ACC runtime does not rely on vendor-specific technology. Instead, MPI-ACC offers a platform-agnostic layer which automatically takes advantage of detected hardware capabilities but is not reliant on them.

The MVAPICH2-X unified communication runtime [27] extends MVAPICH2 to support scheduling of both message passing and PGAS workloads on a single communication runtime. This work therefore enables safer and better performing composition of MPI with various PGAS programming models, but is of course not a general-purpose composability framework. In contrast, many other runtimes exist that serve as the foundation for higher-level communication frameworks that do not easily compose, such as Adaptive MPI over the Charm++ runtime [28], or the many PGAS languages over GASNet [29].

The recent introduction of `target` and `task` directives, as well as the `depend` clause in OpenMP [30] have made composing accelerators and host parallelism using OpenMP possible. A programmer may create dependencies between tasks running on the host and accelerator kernels. While the abstractions offered by OpenMP theoretically enable the composition of any accelerator with host parallelism, in reality this support has only been added for GPUs. However, like HiPER, the higher level abstractions of OpenMP mean that the composability it enables has a much broader scope than most related works.

There have also been several research projects into composing GPUs with host parallelism [15, 31–33]. In general, the approach taken by these works is similar to that taken by work on communication offload: a dedicated GPU management thread is used to schedule work across all GPUs in the system. These works have similar challenges, in that they are usually hard-coded to a single accelerator type and lose a whole OS thread to GPU management.

XKaapi [34] contributes a work-stealing, locality-aware runtime for scheduling tasks with internal parallelism across CPUs and GPUs. This runtime offers automatic data coherency across CPUs and GPUs and automatic load balancing across devices. While the data coherency contributions are less relevant today with the upcoming release of hardware-supported GPU Unified Memory [35], the load balancing contributions of XKaapi would ease programmer burden when combining CPUs and GPUs.

Lithe [36] focuses on composing libraries that use one or more processing units on a shared multi-processor. It proposes APIs which allow these libraries to request and yield cores, relative to a parent in the Lithe scheduler. While this is a scalable and elegant solution, it does require modifications to libraries to make them composable and its scope is limited

to composing systems that share the same computational resource (e.g. an OpenMP host runtime and Intel MKL).

V. Discussion & Future Work

While HiPER's use of modules, generalized work-stealing, and an abstract platform model makes it a general framework, it is important to consider where it might struggle to enable composable components. In particular, we believe HiPER's main challenge is in supporting components that share the CPU with the HiPER runtime itself. For example, supporting composable MKL would require logic in the HiPER runtime for 1) forfeiting CPU cores for the use of MKL, and 2) scheduling MKL on those cores specifically. Indeed, this is the exact challenge that Lithe [36] solves, demonstrating that this type of composition will require modifications to the software components themselves, an undesirable property and something HiPER currently avoids.

An important item to note is the tooling that HiPER enables. Like any unified scheduler, the HiPER runtime is aware of all of the work executing on a system. Hooks have been added to the HiPER runtime which enable programmers to gather statistics on time spent in calls to different modules. HiPER can add high-level, module-specific semantic information on performance bottlenecks. This information was useful in optimizing applications and the HiPER runtime itself.

Ongoing work looks at how inter-component awareness enables inter-component optimization. With the authors of the OpenSHMEM Contexts proposal [37], we are actively exploring how cooperation between the HiPER runtime and a communication runtime enables optimizations.

In addition, there are three types of modules which are left for future work but which we expect to fit well with the HiPER abstractions. First, a HiPER module for checkpointing of application state would enable overlapping of checkpoint I/O with useful application work. Second, we plan continued work on HIPER's data movement APIs, particularly for accelerators, NVRAM, and other future memory technologies. Third, while this section discussed how integrating with host-based numerical libraries would be challenging, supporting accelerator-based numerical libraries would be an important extension.

VI. Conclusions

In conclusion, HiPER is a framework for enabling the composition of a variety of software components, including accelerator libraries, communication libraries, storage libraries, and host parallelism. Using a foundational work-stealing runtime, an abstract platform model, and pluggable software modules HiPER enables unified scheduling of near-arbitrary software components, as well as the expression of dependencies between them.

References

[1] P. Kogge, K. Bergman, S. Borkar, D. Campbell, W. Carson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill *et al.*, "Exascale Computing Study: Technology Challenges in Achieving Exascale Systems," 2008.

[2] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "UPC++: a PGAS Extension for C++," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1105–1114.

[3] OpenSHMEM Specification Committee, "OpenSHMEM," http://openshmem.org/.

[4] N. e. a. Liu, "On the role of burst buffers in leadership-class storage systems," in *012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2012, pp. 1–11.

[5] J. Dongarra, "Report on the sunway taihulight system," *PDF). www. netlib. org. Retrieved June*, vol. 20, 2016.

[6] U. Hanebutte and J. Hemstad, "Isx: A scalable integer sort for co-design in the exascale era," in *Partitioned Global Address Space Programming Models (PGAS), 2015 9th International Conference on*, Sept 2015, pp. 102–104.

[7] M. Adams, "HPGMG 1.0: a Benchmark for Ranking High Performance Computing Systems," 2014.

[8] S. Olivier, J. Huan, J. Liu, J. Prins, J. Dinan, P. Sadayappan, and C.-W. Tseng, "Uts: An unbalanced tree search benchmark," in *Languages and Compilers for Parallel Computing*. Springer, 2007, pp. 235–250.

[9] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group (CUG)*, 2010.

[10] "HiPER Source Code," https://github.com/habanero-rice/hclib/tree/resource_workers.

[11] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. Von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Acm Sigplan Notices*, vol. 40, no. 10. ACM, 2005, pp. 519–538.

[12] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: A generic framework for managing hardware affinities in hpc applications," in *PDP 2010-The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 2010.

[13] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999.

[14] S. Treichler, M. Bauer, and A. Aiken, "Realm: An event-based low-level runtime for distributed memory architectures," in *Proceedings of the 23rd international conference on Parallel architectures and compilation*. ACM, 2014, pp. 263–276.

[15] H. e. a. Kaiser, "HPX: A Task Based Programming

Model in a Global Address Space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 6.

[16] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An api for programming with millions of lightweight threads," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–8.

[17] O. Kowalke, "Boost.Context," Boost.org, 2014, http://www.boost.org/libs/context/.

[18] J. Jose, K. Kandalla, M. Luo, and D. K. Panda, "Supporting hybrid mpi and openshmem over infiniband: Design and performance evaluation," in *2012 41st International Conference on Parallel Processing*. IEEE, 2012, pp. 219–228.

[19] M. Grossman, H. P. Pritchard Jr, Z. Budimlic, and V. Sarkar, "Graph 500 on OpenSHMEM: Using a Practical Survey of Past Work to Motivate Novel Algorithmic Developments," Los Alamos National Laboratory (LANL), Tech. Rep., 2016.

[20] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 712–725.

[21] V. Kumar, Y. Zheng, V. Cavé, Z. Budimlić, and V. Sarkar, "Habaneroupc++: A compiler-free pgas library," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 5.

[22] Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar, "Integrating Asynchronous Task Parallelism with OpenSHMEM," in *OpenSHMEM Workshop*, 2016.

[23] K. Vaidyanathan, D. D. Kalamkar, K. Pamnany, J. R. Hammond, P. Balaji, D. Das, J. Park, and B. Joó, "Improving concurrency and asynchrony in multithreaded mpi applications using software offloading," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, 2015, p. 30.

[24] H. Wang, S. Potluri, D. Bureddy, C. Rosales, and D. K. Panda, "Gpu-aware mpi on rdma-enabled clusters: Design, implementation and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 10, pp. 2595–2605, 2014.

[25] A. M. Aji, L. S. Panwar, F. Ji, K. Murthy, M. Chabbi, P. Balaji, K. R. Bisset, J. Dinan, W.-c. Feng, J. Mellor-Crummey *et al.*, "MPI-ACC: Accelerator-Aware MPI for Scientific Applications," vol. 27, no. 5. IEEE, 2016, pp. 1401–1414.

[26] A. M. Aji, P. Balaji, J. Dinan, W.-c. Feng, and R. Thakur, "Synchronization and Ordering Semantics in Hybrid MPI+GPU Programming," in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE, 2013, pp. 1020–1029.

[27] K. Hamidouche and D. Panda, "MVAPICH2-X: Unified Communication Runtime for Efficient Hybrid MPI+PGAS Programming Models," http://mvapich.cse.ohio-state.edu/static/media/talks/slide/SC16-OSU-PGAS-2.pdf.

[28] C. Huang, O. Lawlor, and L. V. Kale, "Adaptive MPI," in *International workshop on languages and compilers for parallel computing*. Springer, 2003, pp. 306–322.

[29] D. Bonachea, "GASNet Specification, v1.1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-02-1207, Oct. 2002. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2002/5764.html

[30] O. A. R. Board, "OpenMP Application Program Interface Version 4.0 - July 2013," OpenMP.org. [Online]. Available: {http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf}

[31] A. Sbîrlea, Y. Zou, Z. Budimlíc, J. Cong, and V. Sarkar, "Mapping a data-flow programming model onto heterogeneous platforms," in *ACM SIGPLAN Notices*, vol. 47, no. 5. ACM, 2012, pp. 61–70.

[32] J. Bueno, J. Planas, A. Duran, R. M. Badia, X. Martorell, E. Ayguade, and J. Labarta, "Productive programming of gpu clusters with ompss," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 557–568.

[33] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," vol. 23, no. 2. Wiley Online Library, 2011, pp. 187–198.

[34] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*. IEEE, 2013, pp. 1299–1308.

[35] "Inside Pascal: NVIDIA's Newest Computing Platform," https://devblogs.nvidia.com/parallelforall/inside-pascal/.

[36] H. e. a. Pan, "Lithe: Enabling efficient composition of parallel libraries," *Proc. of HotPar*, vol. 9, 2009.

[37] J. Dinan and M. Flajslik, "Contexts: a Mechanism for High Throughput Communication in OpenSHMEM," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*. ACM, 2014, p. 10.