A Pluggable Framework for Composable HPC Scheduling Libraries

Max Grossman\textsuperscript{1}, Vivek Kumar\textsuperscript{2}, Nick Vrvilo\textsuperscript{1}, Zoran Budimlic\textsuperscript{1}, Vivek Sarkar\textsuperscript{1}
\textsuperscript{1}Habanero Extreme Scale Software Research Group, Rice University
\textsuperscript{2}IIIT-Delhi

AsHES 2017 - May 29 2017
Past decade has seen more heterogeneous supercomputers

https://www.top500.org
Majority of Top10 Peak and Achieved GFlop/s has come from heterogeneous machines since 2013

https://www.top500.org
We as a community are very bad at programming heterogeneous supercomputers (even for LINPACK).
How Do We Define Heterogeneity?

For the past decade, “heterogeneous computing” == “GPUs”
• Dealing with GPUs has taught us a lot about software heterogeneity

But heterogeneity is on the rise everywhere in HPC:
• Hardware: memory, networks, storage, cores
• Software: networking libraries, compute libraries, managed runtimes, domain libraries, storage APIs

Depiction of the abstract platform motivating this work
Heterogeneous Programming in Practice

OpenACC
Directives for Accelerators

OpenMP

CUDA

Trilinos

cuDNN

Trilinos

QThreads

OpenMP

pthreads

Open MPI

Open SHMEM
Heterogeneous Programming in Research

Legion: Hide all heterogeneity from user, rely on runtime to map problem to hardware efficiently, implicit dependencies discovered by runtime.

Parsec, OCR: Explicit dataflow model.

HCMPI, HCUPC++, HC-CUDA, HPX: Task-based runtimes that create dedicated proxy threads for managing some external resource (e.g. NIC, GPU).

HiPER: Generalize a task-based, locality-aware, work-stealing runtime/model to support non-CPU resources.

- Retain the appearance of legacy APIs
- Composability, extensibility, compatibility are first-class citizens from the start.
Outline

HiPER Execution & Platform Model

HiPER Use Cases
  • MPI Module
  • Composing MPI and CUDA

Performance Evaluation

Conclusions & Future Work
Outline

HiPER Execution & Platform Model

HiPER Use Cases
• MPI Module
• Composing MPI and CUDA

Performance Evaluation

Conclusions & Future Work
HiPER’s Predecessors

Hierarchical Place Trees
HiPER’s Predecessors

Hierarchical Place Trees w/ GPU

Proxy Thread
HiPER’s Predecessors

Hierarchical Place Trees w/ GPU and OSHMEM
HiPER’s Predecessors

Hierarchical Place Trees w/ GPUs and OSHMEM
HiPER’s Predecessors

GPU → sysmem → MPI → OSHMEM

Proxy Thread

L2

Hierarchical Place Trees w/ GPUs, OSHMEM, MPI
HiPER’s Predecessors

• Simple model makes it attractive for many past research efforts, but…
  • Not scalable software engineering
  • Wasteful use of host resources
  • Not easily extendable to new software/hardware capabilities
HiPER Platform & Execution Model

HiPER Work-Stealing Thread Pool
HiPER Platform & Execution Model

Pluggable Modules

- System
- OSHMEM
- MPI
- CUDA
- etc.

Modules expose user-visible APIs for work creation.

HiPER Work-Stealing Thread Pool

17
HiPER Platform & Execution Model

Pluggable Modules

Platform model gives modules somewhere to place work, thread pool somewhere to find work.

HiPER Platform Model

HiPER Work-Stealing Thread Pool
HiPER Platform & Execution Model

Pluggable Modules

Modules fill in platform model, tell threads the subset of the platform they are responsible for scheduling work on.

HiPER Platform Model

CPU_0  CPU_1  CPU_2  CPU_3  CPU_4

HiPER Work-Stealing Thread Pool
HiPER Platform & Execution Model

Pluggable Modules

System OSHMEM MPI CUDA etc.

HiPER Platform Model

NIC

CPU₀ CPU₁ CPU₂ CPU₃ CPU₄

HiPER Work-Stealing Thread Pool

Modules fill in platform model, tell threads the subset of the platform they are responsible for scheduling work on.
HiPER Platform & Execution Model

Pluggable Modules

System | OSHMEM | MPI | CUDA | etc.

HiPER Platform Model

GPU
CPU₀
CPU₁
CPU₂
CPU₃
CPU₄

HiPER Work-Stealing Thread Pool

CUDA
MPI
OSHMEM
System
HiPER Execution & Platform Model

HiPER Use Cases
• MPI Module
• Composing MPI and CUDA

Performance Evaluation

Conclusions & Future Work
The HiPER core exposes a fundamental C/C++ tasking API.

<table>
<thead>
<tr>
<th>API</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>async([] { S1; })</code></td>
<td>Create an asynchronous task</td>
</tr>
<tr>
<td><code>finish([] { S2; })</code></td>
<td>Suspend calling task until nested tasks have completed</td>
</tr>
<tr>
<td><code>async_at([] { S3; }, place)</code></td>
<td>Create an async. task at a place in the platform model</td>
</tr>
<tr>
<td><code>fut = async_future([] { S4; })</code></td>
<td>Get a future that is signaled when a task completes</td>
</tr>
<tr>
<td><code>async-await([] { S5; }, fut)</code></td>
<td>Create an asynchronous task whose execution is predicated on satisfaction of <code>fut</code>.</td>
</tr>
</tbody>
</table>

Summary of core tasking APIs. The above list is not comprehensive.
MPI Module

Extends HiPER namespace with familiar MPI APIs
• Programmers can use the APIs they already know and love
• Built on 1) an MPI implementation, and 2) HiPER’s core tasking APIs.

Asynchronous APIs return futures rather than MPI_Requests, enabling composability in programming layer with all other future-based APIs:

\[
\text{hiper}::\text{future}_t<\text{void}^*> \ *\text{MPI\_Irecv}/\text{Isend}(...);
\]

Enables non-standard extensions, e.g.:

<table>
<thead>
<tr>
<th>\text{hiper}::\text{future}_t&lt;\text{void}^*&gt; \ <em>\text{MPI_Isend_await}(..., hiper::future_t&lt;\text{void}^</em>&gt; \ *\text{await});</th>
<th>Start an asynchronous send once await is satisfied.</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{hiper}::\text{future}_t&lt;\text{void}^*&gt; \ *\text{MPI_Allreduce_future}(...);</td>
<td>Asynchronous collectives.</td>
</tr>
</tbody>
</table>
Example API Implementation

```cpp
hiper::future_t<void> *hiper::MPI_IsendAwait(..., hiper::future_t<void> *await) {
    // Create a promise to be satisfied on the completion of this operation
    hiper::promise_t<void> *prom = new hiper::promise_t<void>();

    // Taskify the actual MPI_Isend at the NIC, pending the satisfaction of await
    hclib::async_nb_await_at([=] {
        // At MPI place, do the actual Isend
        MPI_Request req;
        ::MPI_Isend(..., &req);

        // Create a data structure to track the status of the pending Isend
        pending_mpi_op *op = malloc(sizeof(*op));
        ...
        hiper::append_to_pending(op, &pending, test_mpi_completion, nic);
    }, fut, nic);

    return prom->get_future();
}
```

Periodic polling function
// Asynchronously process ghost regions on this rank in parallel on CPU
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
kernel_fut = forasync_cuda(..., [] (z) { ... });

// Copy received ghost region to CUDA device
copy_fut = async_copy_await(..., reqs[2], reqs[3], kernel_fut);
Outline

HiPER Execution & Platform Model

HiPER Use Cases
• MPI Module
• Composing MPI and CUDA

Performance Evaluation

Conclusions & Future Work
Task Micro-Benchmarking

Micro-benchmark performance normalized to HiPER on Edison, higher is better.

Experimental Setup

Experiments shown here were run on Titan @ ORNL and Edison @ NERSC.

<table>
<thead>
<tr>
<th>Application</th>
<th>Platform</th>
<th>Dataset</th>
<th>Modules Used</th>
<th>Scaling</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISx</td>
<td>Titan</td>
<td>$2^{29}$ keys per node</td>
<td>OpenSHMEM</td>
<td>Weak</td>
</tr>
<tr>
<td>HPGMG-FV</td>
<td>Edison</td>
<td>log2_box_dim=7, boxes_per_rank=8</td>
<td>UPC++</td>
<td>Weak</td>
</tr>
<tr>
<td>UTS</td>
<td>Titan</td>
<td>T1XXL</td>
<td>OpenSHMEM</td>
<td>Strong</td>
</tr>
<tr>
<td>Graph500</td>
<td>Titan</td>
<td>$2^{29}$ nodes</td>
<td>OpenSHMEM</td>
<td>Strong</td>
</tr>
<tr>
<td>LBM</td>
<td>Titan</td>
<td></td>
<td>MPI, CUDA</td>
<td>Weak</td>
</tr>
</tbody>
</table>
HIPER is low-overhead, no impact on performance for regular applications
HiPER Evaluation – Regular Applications

~2% performance improvement through reduced synchronization from futures-based programming.
HIPER integration improves computation-communication overlap, scalability, load balance.
HiPER used for concurrent (not parallel) programming in Graph500.

Rather than periodic polling, use novel `shmem_async_when` APIs to trigger local computation on incoming RDMA.

Reduces code complexity, hands scheduling problem to the runtime.
Outline

HiPER Execution & Platform Model

HiPER Use Cases
• OpenSHMEM w/o Thread Safety
• OpenSHMEM w/ Contexts

Performance Evaluation

Conclusions & Future Work
HiPER

Working to generalize past work on improving the composability of HPC libraries through tasking. Exploring both improvements at the runtime and API layer.

Drive system requirements using OpenSHMEM, but also currently support composing CUDA, MPI, UPC++.

Future work:
• Continuing work on additional module support (integration with OpenSHMEM contexts)
• Continue to iterate on existing benchmarks
• New application development (Fast Multipole Method)

https://github.com/habanero-rice/hclib/tree/resource_workers