Performance of a Task-Parallel PGAS Programming Model using OpenSHMEM and UCX

Max Grossman\textsuperscript{1} and Howard Pritchard\textsuperscript{2}

\textsuperscript{1}Rice University, \textsuperscript{2}Los Alamos National Laboratory

[ March 28, 2017 ]
Outline

- State of Multi-Threading in OpenSHMEM
- AsyncSHMEM Overview
- API Extensions
- Runtime Implementations
- Performance Evaluation
- Discussion of Contributions & Future Directions
OpenSHMEM Threading Group

Concerned with enabling safe use of OpenSHMEM in a multi-threaded environment.

- OpenSHMEM today is not thread-safe

Bottom-up approach to the general problem of thread safety.

Likely outcome: MPI-like thread safety with OpenSHMEM contexts

```c
for (t = 0; t < nthreads; t++) {
    shmem_ctx_create(0, ctxs + t);
}

#pragma omp parallel
{
    ...
    shmem_ctx_putmem(...,
                        ctxs[omp_get_thread_num()]);
    ...
}
```
OpenSHMEM Threading Group

- Single- or Multi-Threaded App (pthreads, qthreads, OMP, etc.)
- Contexts
- Thread Safe Layer
- OpenSHMEM Runtime
Outline

- State of Multi-Threading in OpenSHMEM
- AsyncSHMEM Overview
- API Extensions
- Runtime Implementations
- Performance Evaluation
- Discussion of Contributions & Future Directions
AsyncSHMEM targets the same problems as the OpenSHMEM threading group.

Look at the problem top-down: how can we make multi-threaded runtimes more OpenSHMEM-aware to improve their use together (productivity and performance).

Encourage asynchrony to protect against variability/latencies in future HPC systems.
AsyncSHMEM Goals and Deliverables

- Explore new APIs at the boundary between OpenSHMEM and tasking APIs
- Develop runtimes to support these extensions and existing APIs:
  1. **Offload Runtime** works with current runtimes, does not rely on thread safety of OpenSHMEM implementation, more opportunities for exploiting asynchrony, especially for new applications (implemented)
  2. **Contexts-Based Runtime** looks ahead to contexts, uses them under the cover to drive the network from multiple threads while minimizing lock contention (in-progress)
AsyncSHMEM Execution Model

PE₀

Task → Task → Task

PE₁

Task → Task → Task

Active message

shm_em_int_put
AsyncSHMEM Under the Covers

Node_0
Worker Threads

AsyncSHMEM extensions
Current OpenSHMEM Runtime

Node_1
Worker Threads

AsyncSHMEM extensions
Current OpenSHMEM Runtime
Open MPI/OSHMEM Runtime

OpenSHMEM API

SPML (put/get)
atomic
sheap

UCX
MXM
Yoda
UCX
MXM
basic
UCX
verbs
mmap/sysV

Network API/Linux shared memory interfaces

UCX components used in this study
Outline

- State of Multi-Threading in OpenSHMEM
- AsyncSHMEM Overview
- **API Extensions**
  - Runtime Implementations
  - Performance Evaluation
- Discussion of Contributions & Future Directions
Creating an asynchronous task --- shmem_task()

```c
void shmem_task(void (*body)(void *), void *data);
    Creates an asynchronous task defined by body (like “begin” construct in Chapel)

void foo(void *data) {// Body of child task
    ...
}

void entrypoint(void *args) {// Body of root task
    shmem_task(foo, NULL);
}

int main(int argc, char** argv) {
    shmem_worker_init(entrypoint, NULL);
}
```
### Join synchronization for parallel tasks --- shmem_task_scope

| void shmem_task_scope_begin();  
<table>
<thead>
<tr>
<th>void shmem_task_scope_end();</th>
</tr>
</thead>
<tbody>
<tr>
<td>Starts and ends a task synchronization scope. Like Chapel’s “sync” construct, <code>shmem_task_scope_end()</code> waits on <em>all</em> tasks created in scope before returning control to the calling task. Task scopes may be nested.</td>
</tr>
</tbody>
</table>

```c
void foo(void *data) {
    shmem_task(bar, NULL);
}

void entrypoint(void *args) {
    shmem_task_scope_begin();
    {
        shmem_task(foo, NULL);
        shmem_task(baz, NULL);
    }
    shmem_task_scope_end(); // Wait for tasks foo, bar, baz
}
```
Futures and Promises

```c
void shmem_satisfy_promise(shmem_promise_t *promise, void *data);
    Store a value into a single-assignment promise.

void shmem_task_await(shmem_future_t *future, void (*body)(void *data), void *data);
    Create an asynchronous task whose execution is predicated on the satisfaction of the specified future.

void producer(void *data) {
    shmem_satisfy_promise((shmem_promise_t *)data, NULL);
}

void consumer(void *data) {
    // Only starts executing after producer satisfies the promise
}

shmem_task_await(shmem_future_for_promise(promise), consumer, NULL);
```
Communication-driven tasks allow remote communication to trigger asynchronous task creation on a PE.

Analogous to existing shmem_wait APIs, but these APIs do not block, and also offer single- and multi-condition variants.
Outline

- State of Multi-Threading in OpenSHMEM
- AsyncSHMEM Overview
- API Extensions
- **Runtime Implementations**
- Performance Evaluation
- Discussion of Contributions & Future Directions
AsyncSHMEM Under the Covers (Recap)

Two implementations of the AsyncSHMEM API:
- **Offload** runtime
- **Contexts** runtime

Both fundamentally based on the same system design: work-stealing, multi-threaded runtime paired with an OpenSHMEM implementation.
- In this case, OpenMPI’s OSHMEM over UCX
Offload Runtime

PE 0

pop
push
Computation Worker

Computation Worker

PE 1

put
lock
barrier_all
get

RDMA

Communication Worker

OpenSHMEM Implementation

Computation Worker

...
Example lifetime of a `shmem_int_put` in the offload runtime:

1. Arguments to the `shmem_int_put` call are wrapped in a task, placed at the communication worker.
2. Calling task is suspended, current worker thread picks up another task.
3. Communication worker eventually picks up `shmem_int_put` task and performs the `shmem_int_put` call.
4. Suspended task is re-inserted into work-stealing runtime.
Contexts Runtime (In-Progress)

PE 0

Worker
SHMEM Context(s)

PE 1

RDMA

Worker
SHMEM Context(s)

Worker
SHMEM Context(s)
Outline

- State of Multi-Threading in OpenSHMEM
- AsyncSHMEM Overview
- API Extensions
- Runtime Implementations
- **Performance Evaluation**
- Discussion of Contributions & Future Directions
Extensions to OpenSHMEM are in part being validated through application benchmarks.

Application focus to date:
• ISx – Distributed integer sort (dataset = up to 2 billion keys per node)
• UTS – Unbalanced tree search (dataset = T1XXL)
• G500 - Distributed breadth first search (dataset = up to 2^27 vertices)

Evaluation shown today performed on LANL Hickok and Rice DAVINCI systems using OpenMPI’s OpenSHMEM implementation over UCX and the AsyncSHMEM Offload runtime.
Experimental Setup

Two clusters were used for these experiments (will also show non-UCX tests on Titan):

DAVINCI cluster at Rice
• Dual socket 2.8 GHz Westmere with 6 cores/socket
• Mellanox ConnectX3 QDR, PCI-e gen2
• RHEL 6.5/OFED 2.2-1

Hickok network technology testbed at LANL
• Dual socket 2.1 GHz Broadwell with 8 cores/socket
• Mellanox ConnectX5 EDR, PCI-e gen3
• 5 36-port EDR switches cabled in fat tree
• RHEL 7.2/MOFED 4.0.1-0
OSU OpenSHMEM micro-benchmark results

UCX_TLS=dc mlx5,sm or UCX_TLS=rc,sm
OSU OpenSHMEM micro-benchmark results

SHMEM multi-rate put throughput

OpenSHMEM Atomics performance

Latency (us)

0 10 20 30 40
Habanero Tasking Micro-Benchmarks
Low overheads of tasking layer yields improvement on hickok, hybrid parallelism improves performance at large scales.
UTS

Titan scaling results

DAVINCI

hickok

Total execution time (s)

Total nodes on hickok (16 cores per node)

Flat OpenSHMEM
AsyncSHMEM
OpenSHMEM + OpenMP
Similar scaling results to 1024 nodes on Titan.
● State of Multi-Threading in OpenSHMEM
● AsyncSHMEM Overview
● API Extensions
● Runtime Implementations
● Performance Evaluation
● Discussion of Contributions & Future Directions
Contributions

Key contributions:
1. Exploration of OpenSHMEM and task-parallel runtimes, tying parallelism and communication together and proposing extensions at the boundary of the two.
2. Two implementations of these extensions.

API extensions for parallelism motivated by Habanero model, with the addition of APIs that connect OpenSHMEM communication with task-parallel execution.
Ongoing Work

Runtime integration with Contexts in collaboration with Jim Dinan, Kayla Seager at Intel.

Continue to iterate on existing benchmarks.

Exploration of algorithmic opportunities opened up by contexts.

New application development (Fast Multipole Method).
Summary of New APIs

Environment
• shmem_worker_init
• shmem_my_worker
• shmem_n_workers

Fork-join tasks
• shmem_task
• shmem_parallel_for
• shmem_task_scope_begin
• shmem_task_scope_end

Futures and promises
• shmem_satisfy_promise
• shmem_future_wait
• shmem_task_future
• shmem_task_await

Communication-driven tasks
• shmem_int_task_when
• shmem_int_task_when_any
Environment APIs --- Hello World Example

```c
void shmem_worker_init(void (*entrypoint)(void *), void *data);
    Initializes both the OpenSHMEM (using shmem_init and shmem_finalizer) and work-stealing runtimes. entrypoint is the root task of the PE. The number of worker threads created is configurable by environment variable.

int shmem_my_worker();
    Returns a unique ID for the calling thread.

int shmem_n_workers();
    Returns the number of threads in the thread pool for the calling PE.

void entrypoint(void *args) {
    printf("This is thread %d, PE %d\n", shmem_my_worker(), shmem_my_pe());
}

int main(int argc, char** argv) {
    shmem_worker_init(entrypoint, NULL);
}
```
Creating a range of parallel tasks --- shmem_parallel_for()

```c
void shmem_parallel_for(int lower_bound, int upper_bound,
                         void (*body)(int, void *), void *data);
Efficiently creates a batch of tasks, one for each integer in the range [lower_bound, upper_bound). There is no implicit synchronization at the end of a call to shmem_parallel_for.

void foo(int iter, void *data) {
    printf("Hello from parallel iteration %d\n", iter);
}

void entrypoint(void *args) { // Create 100 tasks with indices 0..99
    shmem_parallel_for(0, 100, foo, NULL);
}

int main(int argc, char** argv) {
    shmem_worker_init(entrypoint, NULL);
}
```
Futures and Promises

void entrypoint(void *args) {
    shmepromise_t *promise = shmem_create_promise();
    shmem_future_t *future = shmem_future_for_promise(promise);
}

Create promise and future objects (akin to std::future and std::promise in C++).
Futures and Promises

```c
void shmem_satisfy_promise(shmem_promise_t *promise, void *data);
    Store a value into a single-assignment promise.

void *shmem_future_wait(shmem_future_t *future);
    Wait for a future to be satisfied, and return its value.

void producer(void *data) {
    shmem_satisfy_promise((shmem_promise_t *)data, NULL);
}

void consumer(void *data) {
    void *result = shmem_future_wait((shmem_future_t *)data);
}

shmem_task(producer, promise);
shmem_task(consumer, shmem_future_for_promise(promise));
```
void load_balancer(void *data) {
    // Load balancing logic here, based on updated info from remote PE
    ...
    // Re-register load balancer to handle new updates from PE 1
    shmem_int_task_when(...);
}

shmem_int_task_when(remote_pe_load, SHMEM_CMP_NE, *remote_pe_load, load_balancer, NULL);
...

// Called periodically to update PE 0 with info for distributed work stealing
shmem_int_put(remote_pe_load, local_pe_load, 1, 0);