Integrating Asynchronous Task Parallelism with OpenSHMEM

Max Grossman, Vivek Kumar, Zoran Budimlic, Vivek Sarkar

Habanero Extreme Scale Software Research Group

Rice University

Outline

- <u>OpenSHMEM Threading Group</u>
- 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, motivated by growing benefits of hybrid programming.

Takes a bottom-up approach to the general problem of thread safety.

Three main proposals to date: MPI-like thread safety, Thread Registration, Contexts

OpenSHMEM Threading Group

Approach	Proposer	Summary	API Changes?	Performance Impact	Programmability (Subjective)
MPI-like	Manjunath Gorentla Venkata	Different modes of thread-safe or -unsafe execution, i.e. MPI_Init_thread	Small	Overheads from contention management	No change
Thread Registration	Monika ten Bruggencate et al. (OpenSHMEM 2014)	Any thread to make OSHMEM calls must be registered with the OSHMEM runtime. Runtime manages thread-local data automatically.	API extensions	Negligible	Programmer must remember register/ deregister calls
Contexts	James Dinan et al. (PGAS 2014)	Contexts encapsulate OpenSHMEM resources, thread-safe access to contexts is user responsibility	Major	Negligible	More explicit management of contexts

Outline

- OpenSHMEM Threading Group
- <u>AsyncSHMEM Overview</u>
- API Extensions
- Runtime Implementations
- Performance Evaluation
- Discussion of Contributions & Future Directions

AsyncSHMEM Overview

AsyncSHMEM targets the same problems as the thread-safety proposals.

Looking at the problem top-down: how can we support OpenSHMEM in a threading runtime to improve their use together (productivity and performance).

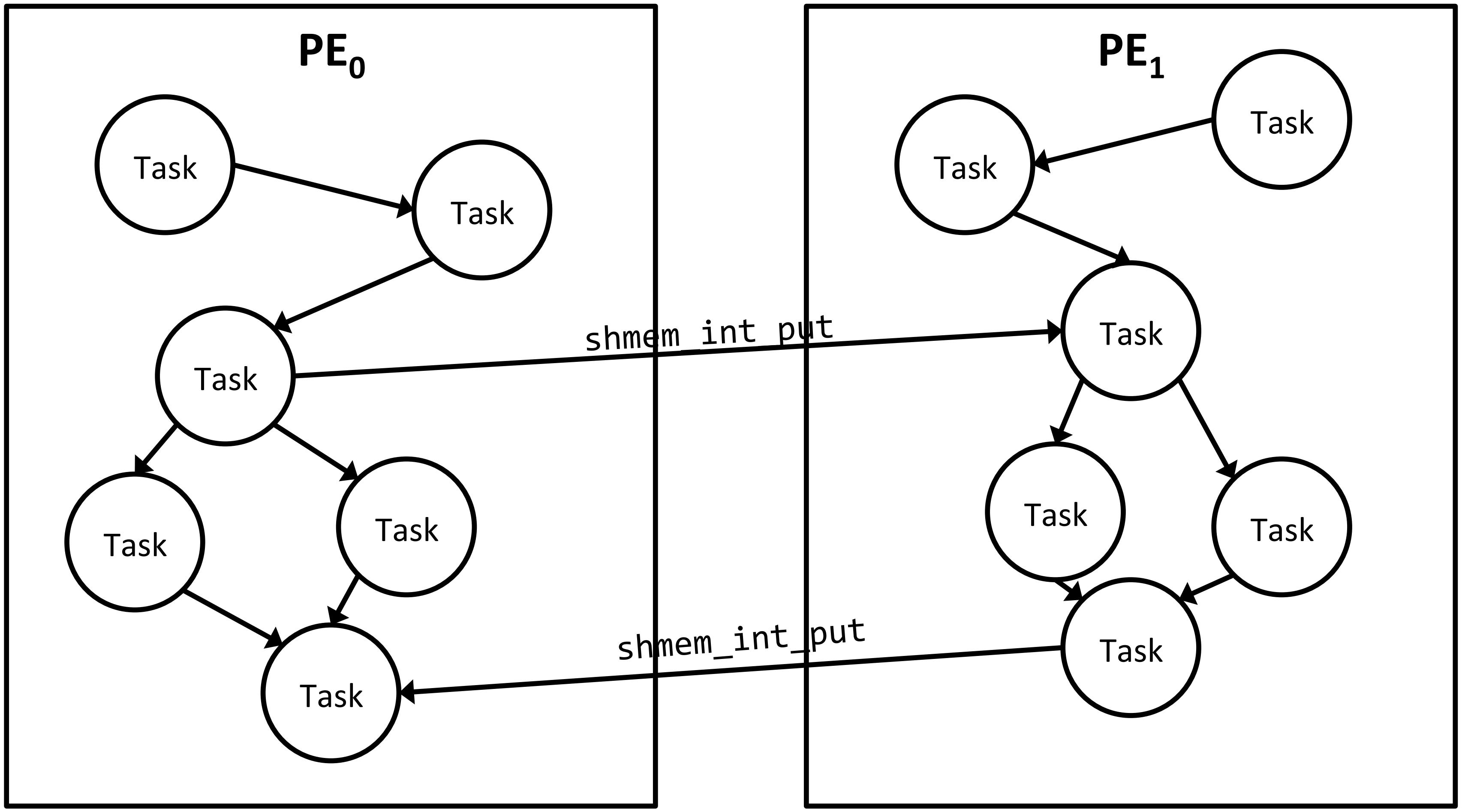
Encourage asynchrony to protect against variability in future HPC systems.

AsyncSHMEM Goals and Approach

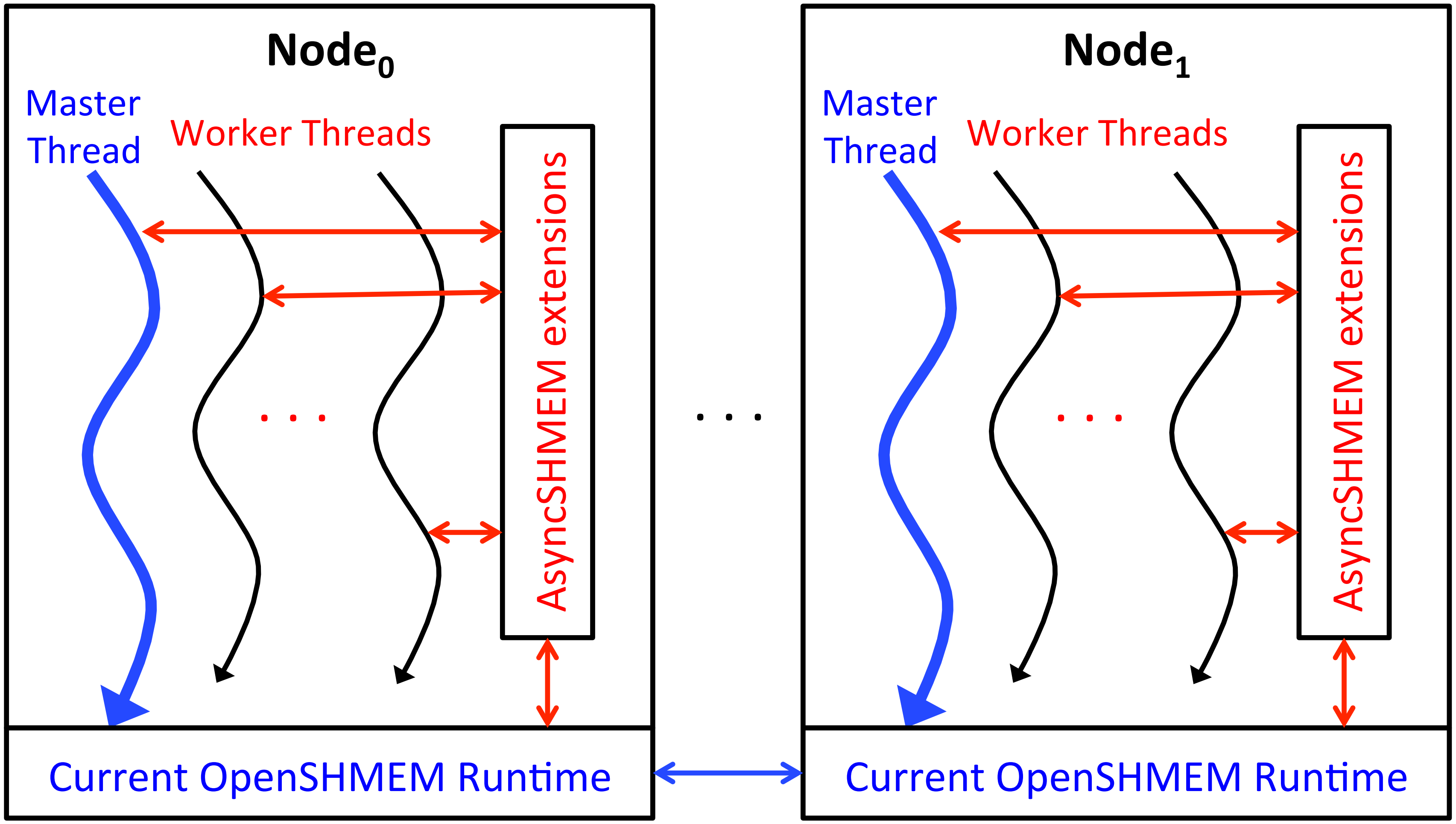
- Extend OpenSHMEM API with unified capabilities for asynchronous computational and communication tasks
- Develop two runtimes to support these API extensions:

1. *Fork-Join Runtime* constrains programming model in use of API extensions to minimize impact on existing OpenSHMEM applications (conservative option)
2. *Offload Runtime* offers a more flexible programming model, and more opportunities for exploiting asynchrony, especially for new applications (forward-looking option)

AsyncSHMEM Execution Model



AsyncSHMEM Under the Covers



Consistency with OpenSHMEM Semantics

Ordering, collectivity, atomicity semantics in AsyncSHMEM remain consistent with existing OpenSHMEM specification.

Collectives: Each PE must take part, only one task per PE, and programmer must ensure ordering (e.g. `shmem_barrier_all`).

Ordering: Memory ordering guarantees based on task graph (e.g. `shmem_fence`).

Atomicity: Atomic with respect to other local threads and other PEs (e.g. `shmem_add`).

Outline

- OpenSHMEM Threading Group
- AsyncSHMEM Overview
- <u>API Extensions</u>
- Runtime Implementations
- Performance Evaluation
- Discussion of Contributions & Future Directions

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_create_promise
- shmem_future_for_promise
- 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

```
void shmem_worker_init(void (*entrypoint)(void *), void *data);
```

Initializes *both* the OpenSHMEM (using shmem_init and shmem_finalize) 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 an asynchronous task: shmem_task()

```
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);
}
```

Creating a range of parallel tasks: shmem_parallel_for()

```
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);
}
```

API Extensions: Join synchronization for parallel tasks

```
void shmem_task_scope_begin();
void shmem_task_scope_end();
```

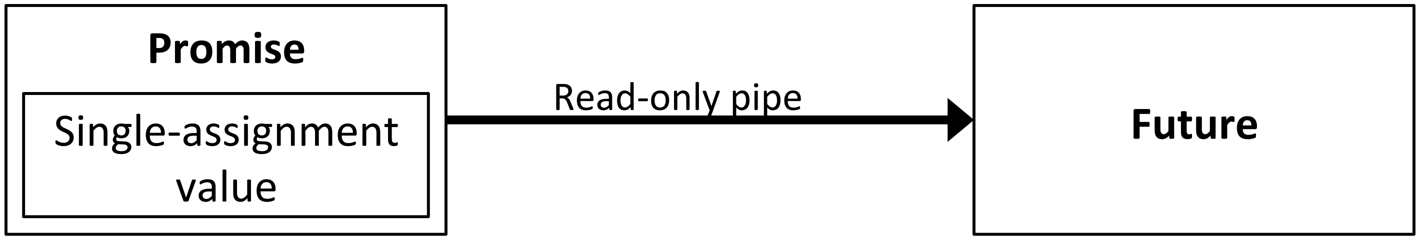
Starts and ends a task synchronization scope. Like Chapel's "sync" construct, `shmem_task_scope_end()` waits on *all* tasks created in scope before returning control to the calling task. Task scopes may be nested.

```
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
}
```

API Extensions: Futures and Promises

```
shmem_promise_t *shmem_create_promise();
shmem_future_t *shmem_future_for_promise(shmem_promise_t *promise);
```

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



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

API Extensions: Futures and Promises (contd)

```
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));
```

API Extensions: Futures and Promises (contd)

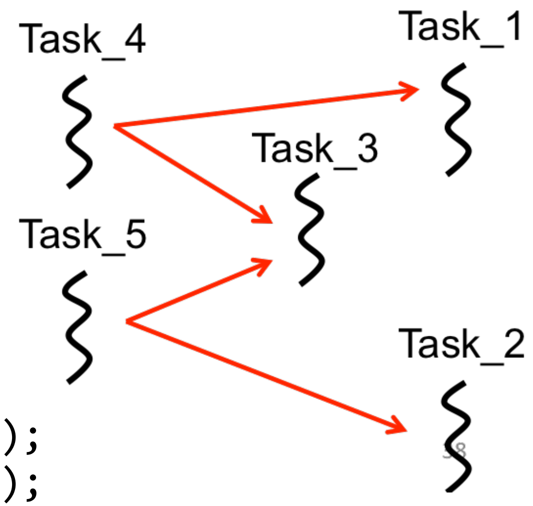
```
void shmem_task_await(shmem_future_t *future, void (*body)(void *data), void *data);
```
Create an asynchronous task whose execution is predicated on the satisfaction of the specified future.

```
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_async(producer, promise);

shmem_task_await(shmem_future_for_promise(promise), consumer, NULL);
```

Example of DAG parallelism using futures

Futures enable more complex dependency graphs than fork-join tasks

```
void task_4(void *task4_prom) {
    // some computation
    shmem_satisfy_promise((shmem_promise_t *)task4_prom);
}
void task_5(void *task5_prom) {
    // some computation
    shmem_satisfy_promise((shmem_promise_t *)task5_prom);
}

shmem_task_await(task_1, args, shmem_future_for_promise(task4_prom));
shmem_task_await(task_2, args, shmem_future_for_promise(task5_prom));
shmem_task_await(task_3, args, shmem_future_for_promise(task4_prom),
        shmem_future_for_promise(task5_prom));
shmem_task(task_4, task4_prom);
shmem_task(task_5, task5_prom);
```



API Extensions: Communication-Driven Tasks

```
void shmem_int_task_when(int *ivar, int cond, int value,
        void (*body)(void *), void *data);
```

Create an asynchronous task when the specified condition is satisfied on the specified location in the symmetric heap. Analogous to `shmem_int_wait_until`, except that this call never blocks.

```
void shmem_int_task_when_any(int **ivars, int cond, int *values,
        void (*body)(void *), void *data);
```

Same as `shmem_int_task_when`, but allows waiting for any of multiple conditions.

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.

Example with Communication-Driven Tasks

PE 0

```
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);
...
```

PE 1

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

Outline

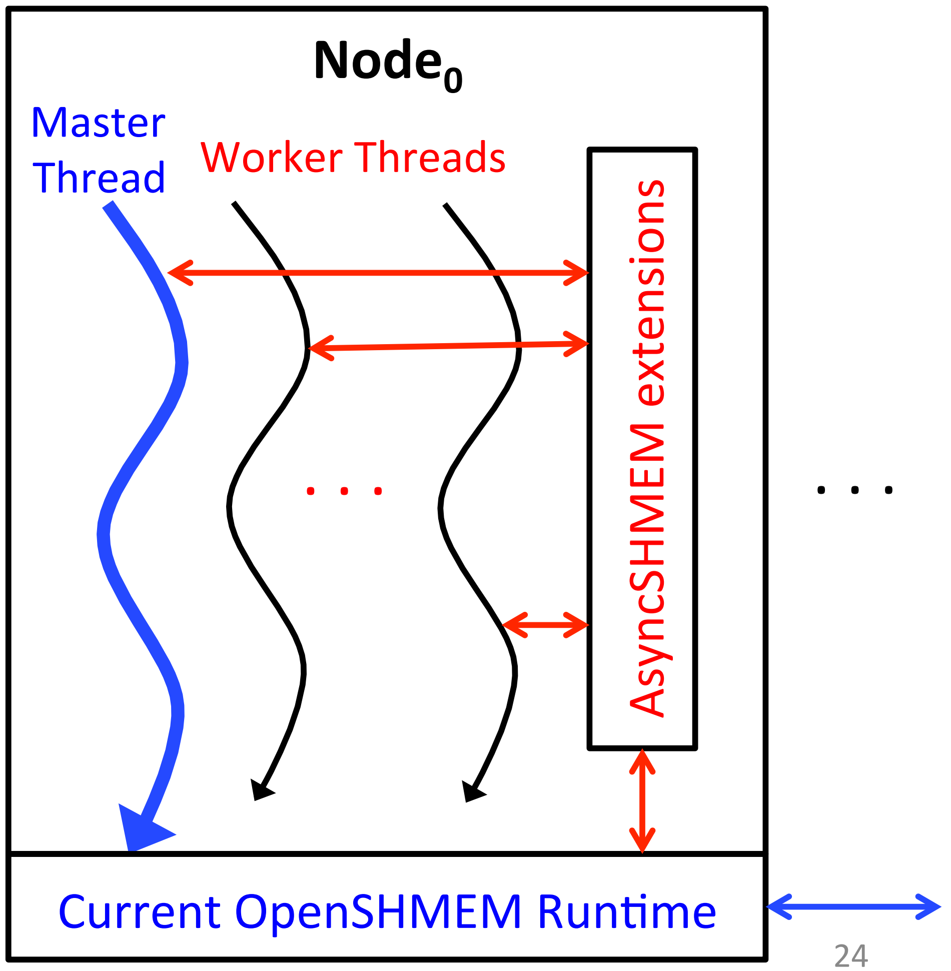
- OpenSHMEM Threading Group
- AsyncSHMEM Overview
- API Extensions
- <u>Runtime Implementations</u>
- Performance Evaluation
- Discussion of Contributions & Future Directions

AsyncSHMEM Under the Covers (Recap)

We have two implementations of the AsyncSHMEM API:

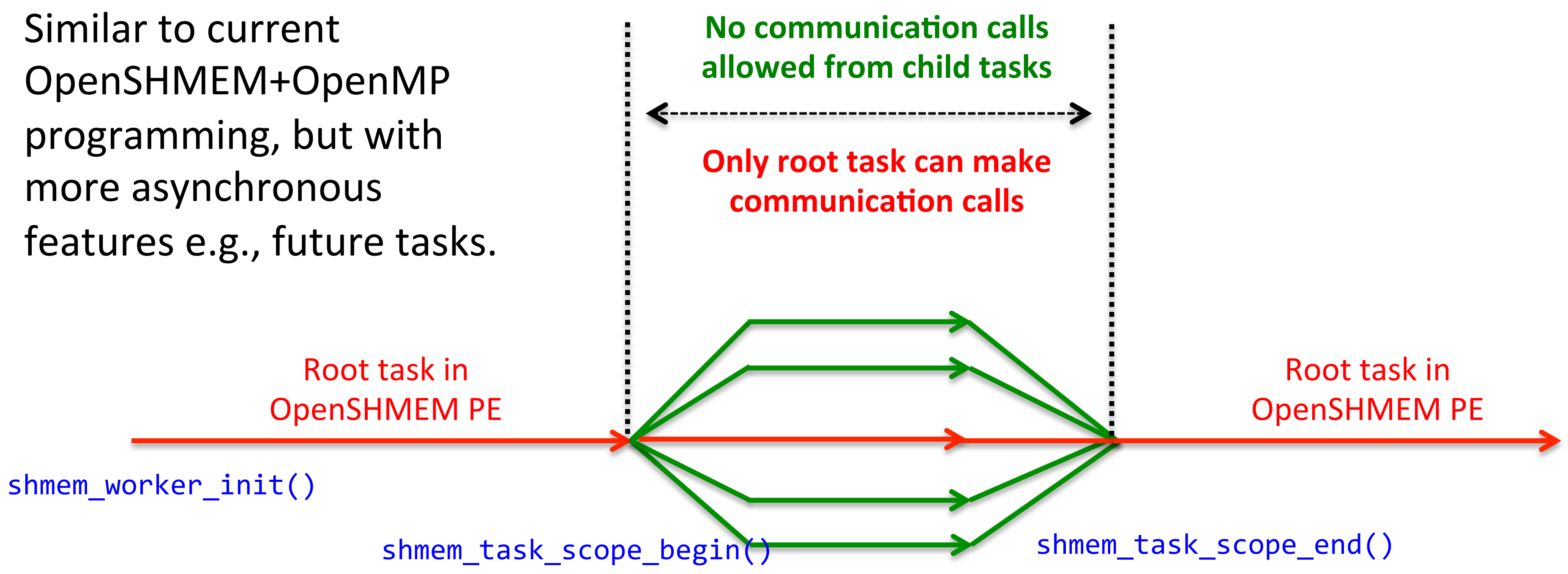
- **Fork-Join** runtime
- **Offload** runtime

Both fundamentally based on the same system design: work-stealing, multi-threaded runtime paired with an OpenSHMEM implementation.

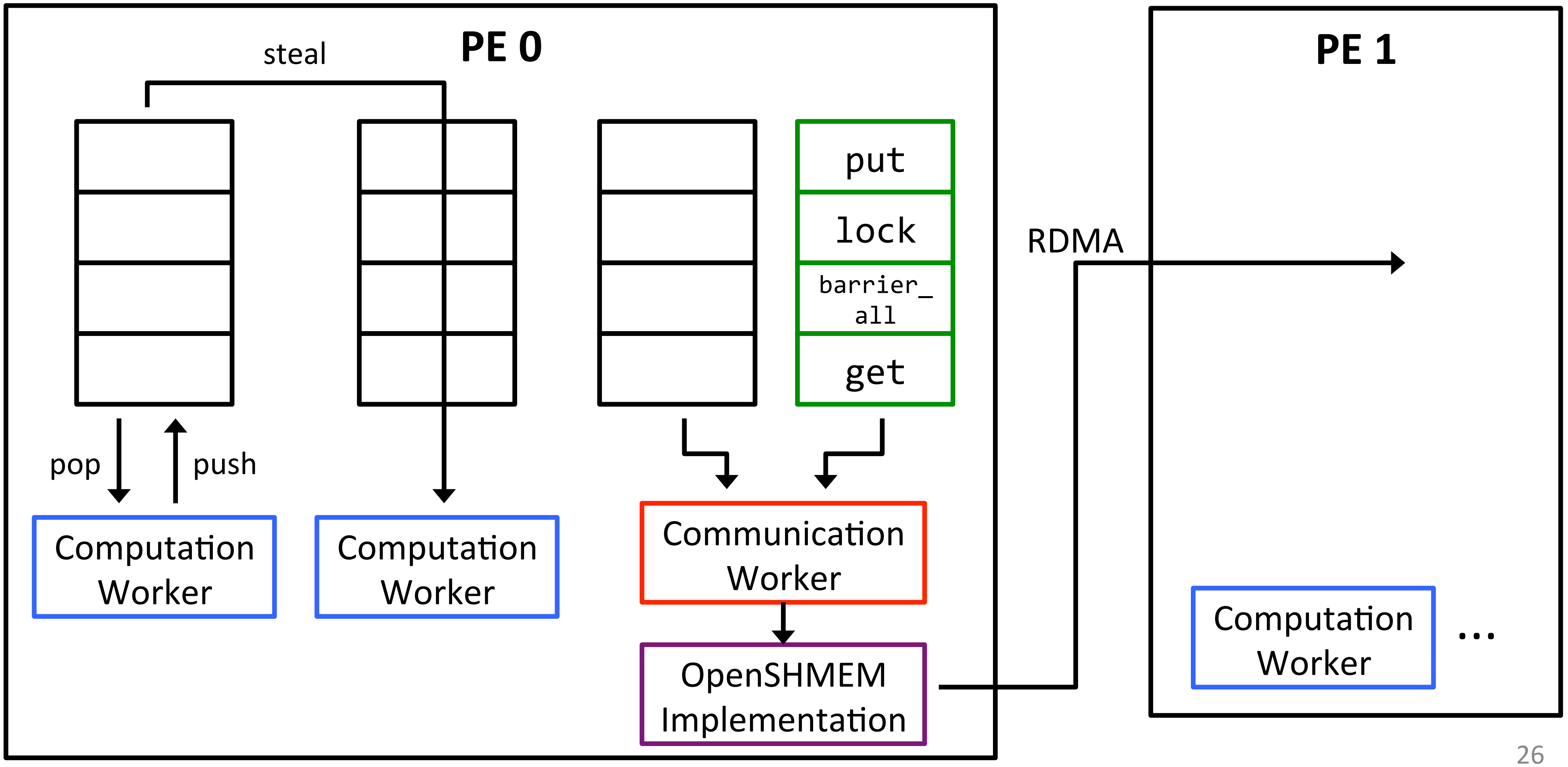


Fork-Join Runtime

Similar to current OpenSHMEM+OpenMP programming, but with more asynchronous features e.g., future tasks.


No communication calls allowed from child tasks
Only root task can make communication calls
Root task in OpenSHMEM PE
Root task in OpenSHMEM PE
shmem_worker_init()
shmem_task_scope_begin()
shmem_task_scope_end()

Offload Runtime



Offload Runtime (cont'd)

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.

Outline

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

Application Benchmarks

Extensions to OpenSHMEM are in part being validated through application benchmarks.

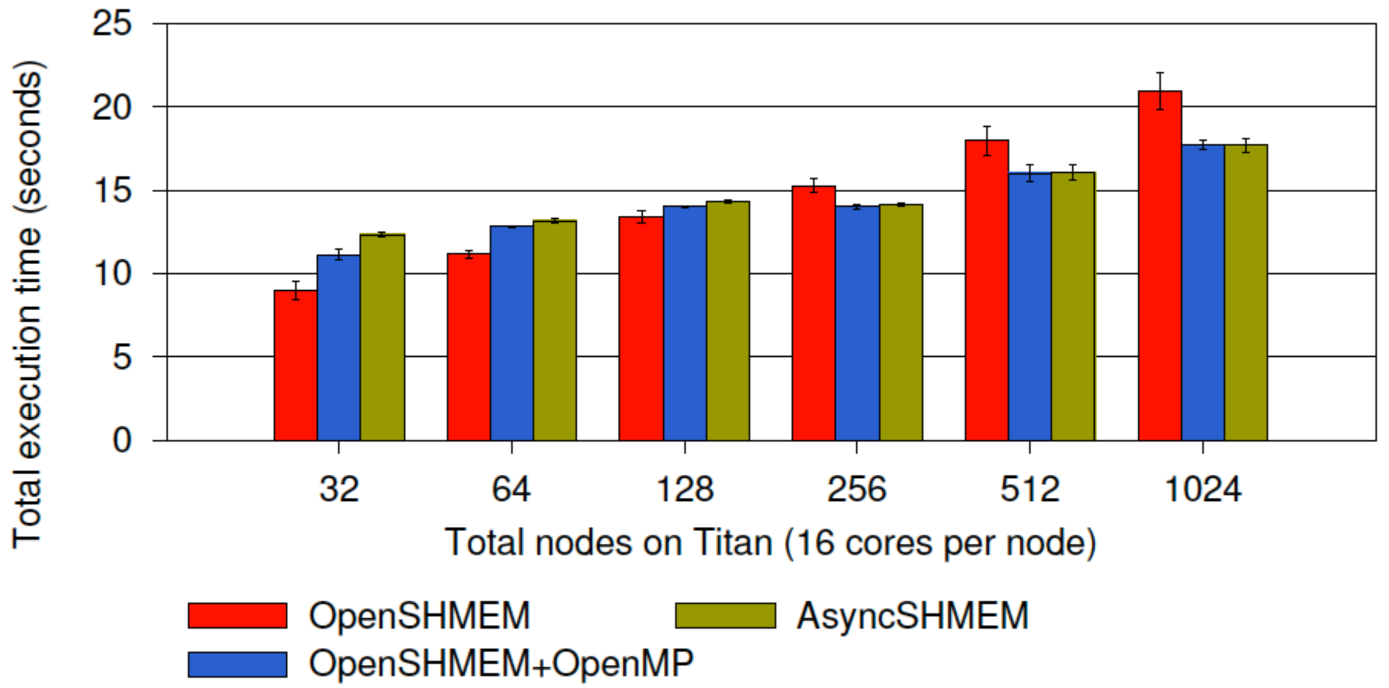
Application focus to date:

- UTS – Unbalanced tree search
- ISx – Distributed integer sort
- Graph500 – Distributed graph search (in-progress)

Evaluation shown today performed on Titan system at ORNL.

ISx

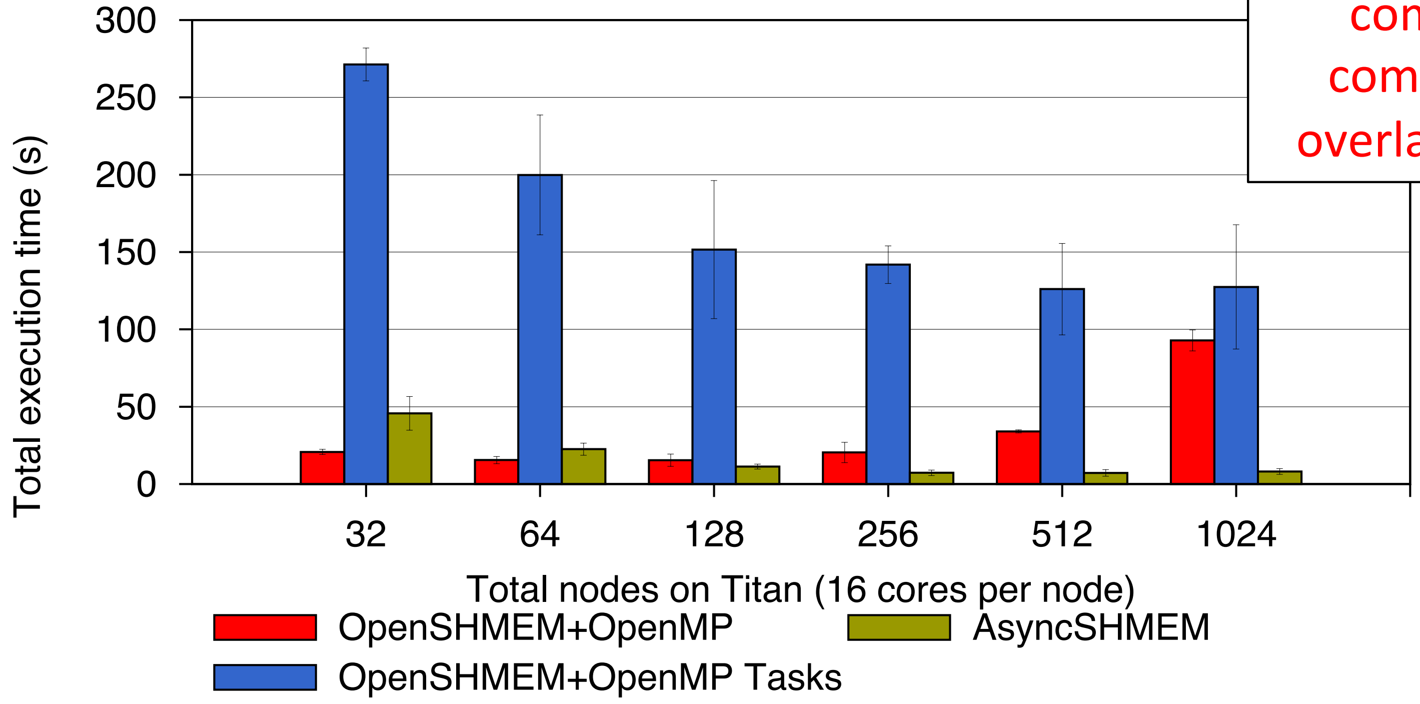
ISx benchmark – Fork-Join approach, weak scaling



(a) Total execution time

UTS results

UTS (T1XXL) – Offload approach



AsyncSHMEM integration improves computation-communication overlap, scalability

Outline

- OpenSHMEM Threading Group
- AsyncSHMEM Overview
- API Extensions
- Runtime Implementations
- Performance Evaluation
- <u>Discussion of Contributions & Future Directions</u>

Contributions

Key contributions:

1. OpenSHMEM extensions for expressing parallelism, tying parallelism and communication together.
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.

Fork-Join runtime minimizes changes to system behavior of existing OpenSHMEM applications, constrains programming model.

Offload runtime maximizes integration, with more flexible programming model.

Future Work

- Explore integration of asynchronous APIs with OpenSHMEM collectives
- Explore extending AsyncSHMEM API, runtime for distributed load balancing.
- Remote tasking (see Sid's talk at 11:30am)
- Explore additional applications in AsyncSHMEM (suggestions welcome), explore algorithmic improvements to current applications enabled by OpenSHMEM
- Continue to improve performance analysis methodology and bottleneck quantification, build more performance metrics into AsyncSHMEM

Acknowledgements

Los Alamos
NATIONAL LABORATORY
EST. 1943

DEPARTMENT OF DEFENSE
UNITED STATES OF AMERICA