Graph500 on OpenSHMEM: Using A Practical Survey of Past Work to Motivate Novel Algorithmic Developments

Max Grossman[1], Howard Pritchard[2], Zoran Budimlic[1], Vivek Sarkar[1]
[1]Rice University, [2]Los Alamos National Laboratory

Outline

Background:
- The OpenSHMEM Programming Model
- Graph500 and the BFS Kernel
- Existing Implementations

Methods:
- Flat OpenSHMEM Implementations
- Hybrid OpenSHMEM Implementations w/ OpenSHMEM Contexts

Evaluation
- Comparison to Reference MPI Implementations

Conclusions

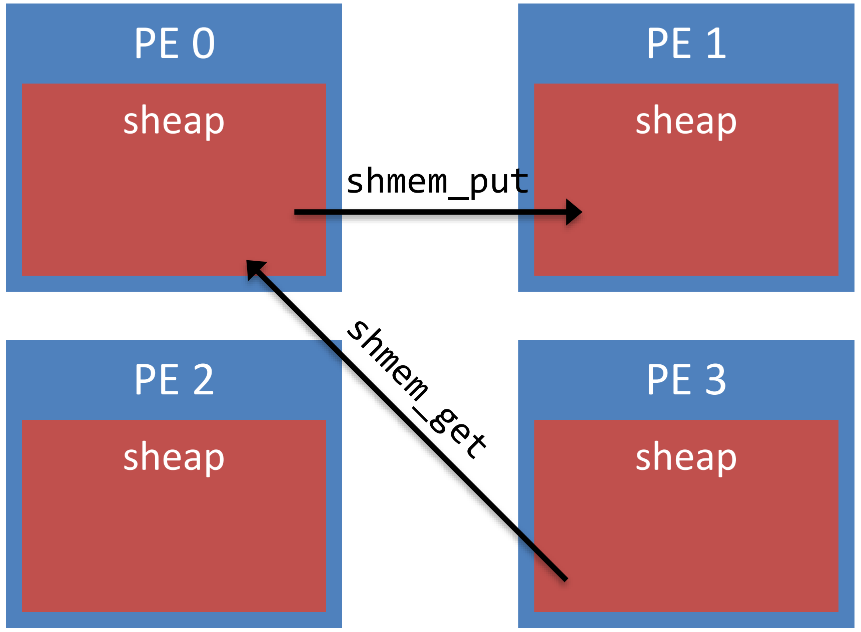
OpenSHMEM

PGAS, SPMD programming model

Core abstraction = symmetric heap and symmetric allocations

Supports efficient communication for applications with small to medium sized messages.

- RDMA, remote atomics, collectives, point-to-point sync.
- Fences and quiets



OpenSHMEM

PE 0

```
int *a = shmem_malloc(sizeof(*a));
int *b = shmem_malloc(sizeof(*b));

*b = 43;

// Send 42 to the sym var a on PE 1
shmem_int_p(a, 42, 1);

// Collective barrier
shmem_barrier_all();
```

PE 1

```
int *a = shmem_malloc(sizeof(*a));
int *b = shmem_malloc(sizeof(*b));

// Wait for receipt of 42 in a
shmem_int_wait_until(a,
    SHMEM_CMP_EQ, 42);

// Retrieve the value 43 from PE 0
shmem_int_g(b, 0);

// Collective barrier
shmem_barrier_all();
```

OpenSHMEM Contexts

Enables creation of separate logical endpoints into the network.

```
#pragma omp parallel
{
    shmem_ctx_t my_ctx;
    shmem_ctx_create(SHMEM_CTX_PRIVATE, &ctx);
    shmem_ctx_int_put(..., my_ctx);
}
```

Several benefits:

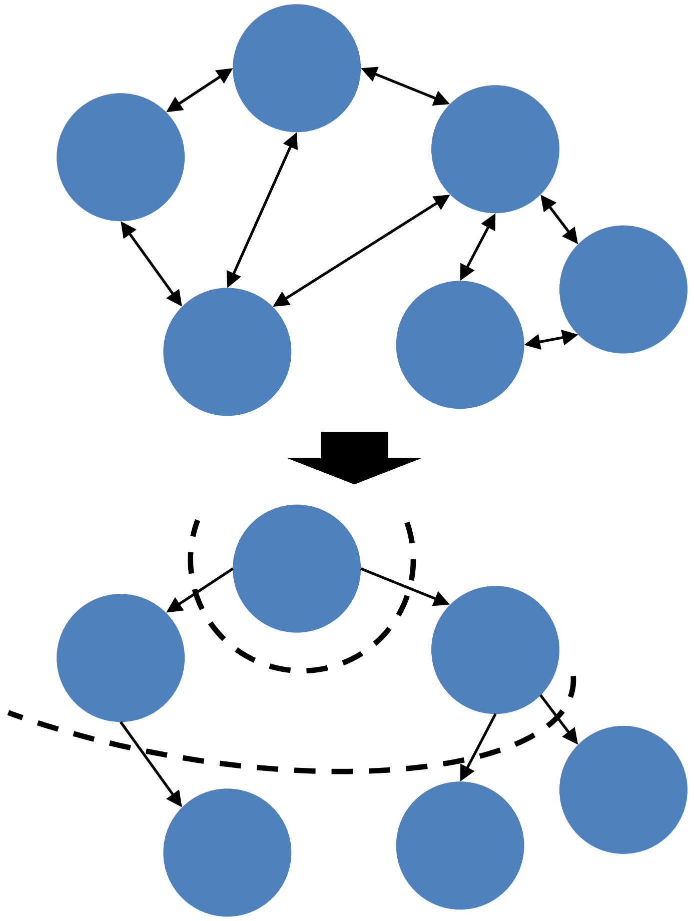
- Independent memory fences, quieting on different contexts
- Relaxation of thread safety on individual contexts reduces overheads
- Offers hints to runtime as to how to partition network resources
- Significant performance benefits at small to medium packet sizes

Graph500

Open benchmark for high performance computing, emphasis on irregular and fine grain data access/computation.

Researchers commonly focus on the BFS kernel.

Distributed reference implementations are MPI-based, but Graph500's small/medium messages are natural fit for OpenSHMEM.



MPI Reference Implementations

Simple: naïve but easily understood implementation of distributed BF

- Message queue-based
- Scattered `MPI_Test` calls
- Does not scale.

One-Sided: algorithmically similar to Simple but uses MPI one-sided APIs.

Replicated: Communicates via all-gathers of vertex bit masks

- Can profitably use multi-threading.

```
for each vertex in current wavefront:

  for each edge on current vertex:

    owner = VERTEX_OWNER(edge.tgt)

    if  owner == my_rank:
      if not is_visited(edge.tgt):
        set_visited(edge.tgt)
        update predecessor map
        insert in next wavefront’s queue
    else:

      add edge.tgt to buffer[owner]

      if buffer[owner].is_full:
        send buffer[owner] to owner
```

Iterate over local vertices in wavefront

Save locally

Buffer and send remotely

Research Implementations

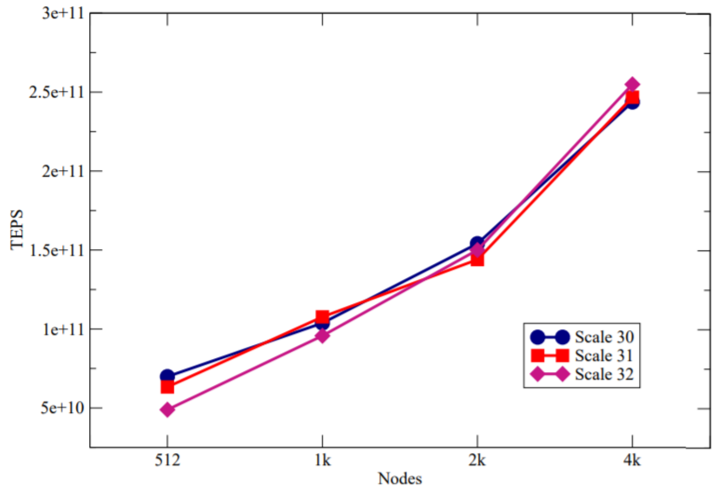
MPI Tuned: From "Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines"

- Does not compile for many rank counts
- Performed comparably to Replicated

D'Azevedo et al (2014) performed a port of One-Sided to OpenSHMEM; emphasized missing capabilities in OpenSHMEM

Jose et al (2013) ported Simple to OpenSHMEM, with RDMA extensions

BGQ strong scaling



Contributions

Survey of open source past work on Graph500's BFS kernel in MPI and OpenSHMEM.

Develop several new open source OpenSHMEM implementations, flat and hybrid (OpenSHMEM + OpenMP).

- Use OpenSHMEM contexts for hybrid implementations

Evaluate new OpenSHMEM implementations against reference MPI implementations out to 1,024 Edison nodes.

Outline

Background:

- The OpenSHMEM Programming Model
- Graph500 and the BFS Kernel
- Existing Implementations

Methods:

- Flat OpenSHMEM Implementations
- Hybrid OpenSHMEM Implementations w/ OpenSHMEM Contexts

Evaluation

- Comparison to Reference MPI Implementations

Conclusions

Flat OpenSHMEM Implementations

Bitvector

- Found MPI Replicated performed best out of reference implementations
- Ported Replicated to OpenSHMEM, exchanged bitvectors with all-gathers
- Ran into issues with Cray OpenSHMEM all-gather performance

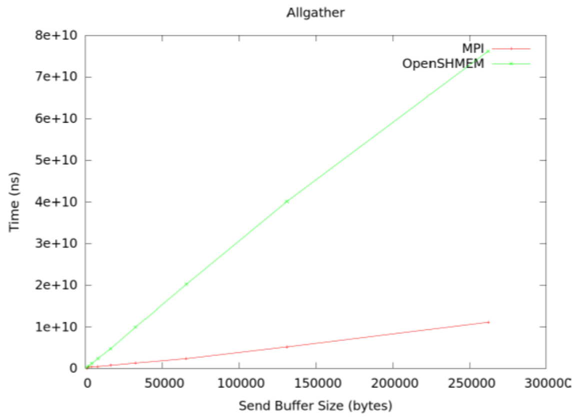


Figure 1: Time to complete 10 all-gather operations using Cray MPICH and Cray SHMEM with various send buffer sizes. Experiments performed on the Edison supercomputer.

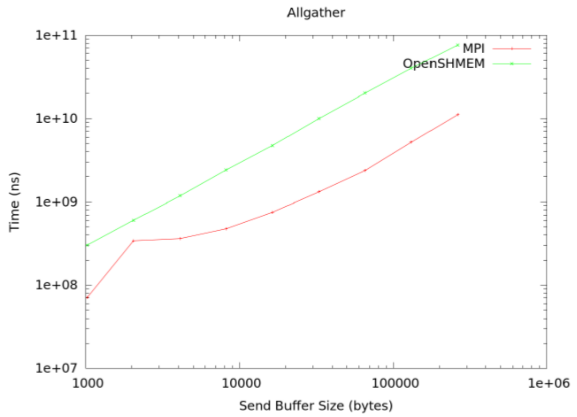


Figure 2: Time to complete 10 all-gather operations using Cray MPICH and Cray SHMEM with various send buffer sizes. Experiments performed on the Edison supercomputer. Axes are in log scale.

Flat OpenSHMEM Implementations

Concurrent-Fence

- Inspired by work in Jose et al (2013)
- Computation-communication overlap
- Fixes packet delivery ordering, but requires an expensive fence

```
for each vertex in current wavefront:

  for each edge on vertex:
    if not is_visited(edge.tgt):
      target_pe = owner_pe(edge.tgt)

      send_buffer[target_pe] += edge.tgt
      if send_buffer[target_pe].is_full():
        shmem_put_nbi(send_buffer[target_pe].body)

        shmem_fence()

        shmem_put_nbi(send_buffer[target_pe].header)

  check for incoming send buffers and process them
```

Flat OpenSHMEM Implementations

Concurrent-Hash

- Incremental improvement on Concurrent-Fence
- Rather than using fence to ensure safe packet delivery, include hash of packet header/body in packet and check it on receiving end
- Explored several hashing algorithms (CRC32, MurmurHash, CityHash); found CityHash performed best for our data
  - Profiling shows that PEs spend ~1% of execution time hashing

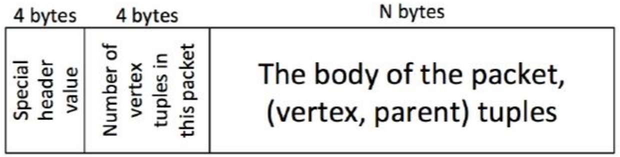

4 bytes
4 bytes
N bytes
Special header value
Number of vertex tuples in this packet
The body of the packet, (vertex, parent) tuples

Figure 3: Format of a send buffer in the Concurrent-Fence implementation.

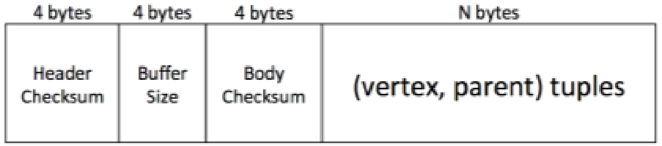


Figure 4: Format of a send buffer in the Concurrent-Hash implementation.

Flat OpenSHMEM Implementations

Wavefront-Puts

- On each wavefront, traverse all local vertices and perform remote PUTs to signal their neighbors

- Upside: Clean implementation

- Downside: each PUT is 4 bytes, lots of very fine grain network traffic

```
// Traverse vertices owned by this PE
for (vertex = local_vertex_start;
    vertex < local_vertex_end; vertex++) {

  // Vertex being visited in the current wavefront
  if (in_curr_wavefront(vertex)) {
    set_visited(vertex);

    // Iterate over the current vertex's neighbors
    for (nbr = nbor_start(vertex);
        nbr < nbor_end(vertex); nbr++) {

      // If this neighbor hasn't been visited,
      // notify its owner with a PUT
      if (!visited(nbr)) {
        shmem_putmem(&preds[local_offset(nbr)],
          &vertex, sizeof(vertex), owner_pe(nbr));
        set_visited(nbr);
      }
    }
  }
}
```

Flat OpenSHMEM Implementations

Wavefront-Atomics

- Incremental improvement on Wavefront-Puts, signal using a bitwise OR instead of PUT

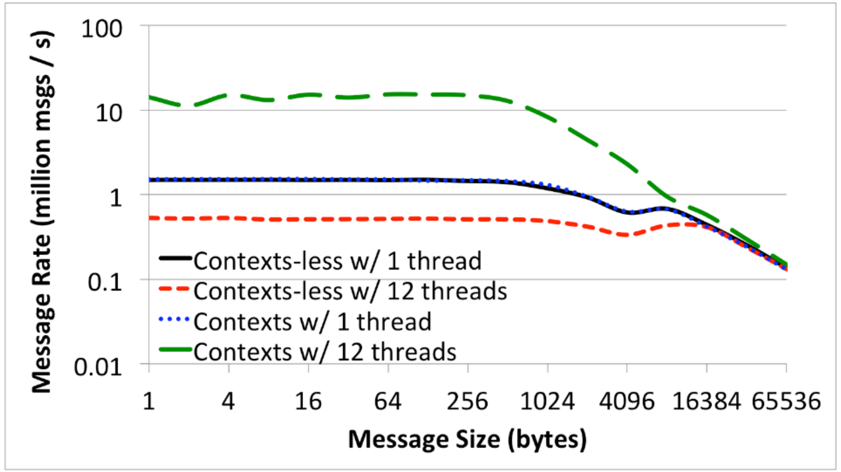
- Enables aggregation of multiple remote signals into a single atomic RDMA, reduced network traffic

```
// Traverse vertices owned by this PE
for (vertex = local_vertex_start;
    vertex < local_vertex_end; vertex++) {

  // Vertex being visited in the current wavefront
  if (in_curr_wavefront(vertex)) {
    set_visited(vertex);

    // Iterate over the current vertex's neighbors
    for (nbr = nbor_start(vertex);
        nbr < nbor_end(vertex); nbr++)
      // If this neighbor hasn't been visited
      if (!visited(nbr)) {
        curr_signals[owner_pe] |= BIT_MASK(nbr);
        set_visited(nbr);
      }
  }
}

for each PE
  shmemx_ctx_uint64_atomic_or(...);
```

Hybrid OpenSHMEM Implementations

Wavefront-Puts/Atomics are simple, but poor in performance.

OpenSHMEM contexts enable multiple threads to drive the network independently -> increased utilization at small and medium packet sizes.



PUT

```
// Traverse vertices owned by this PE
#pragma omp parallel for
for (vertex = local_vertex_start;
    vertex < local_vertex_end; vertex++) {

  ...
   shmem_ctx_putmem(&preds[local_offset(nbr)],
     &vertex, sizeof(vertex), owner_pe(nbr),
     ctx);
}
```

Outline

Background:
- The OpenSHMEM Programming Model
- Graph500 and the BFS Kernel
- Existing Implementations

Methods:
- Flat OpenSHMEM Implementations
- Hybrid OpenSHMEM Implementations w/ OpenSHMEM Contexts

Evaluation
- Comparison to Reference MPI Implementations

Conclusions

Evaluation Methodology

Experiments run on NERSC's Edison

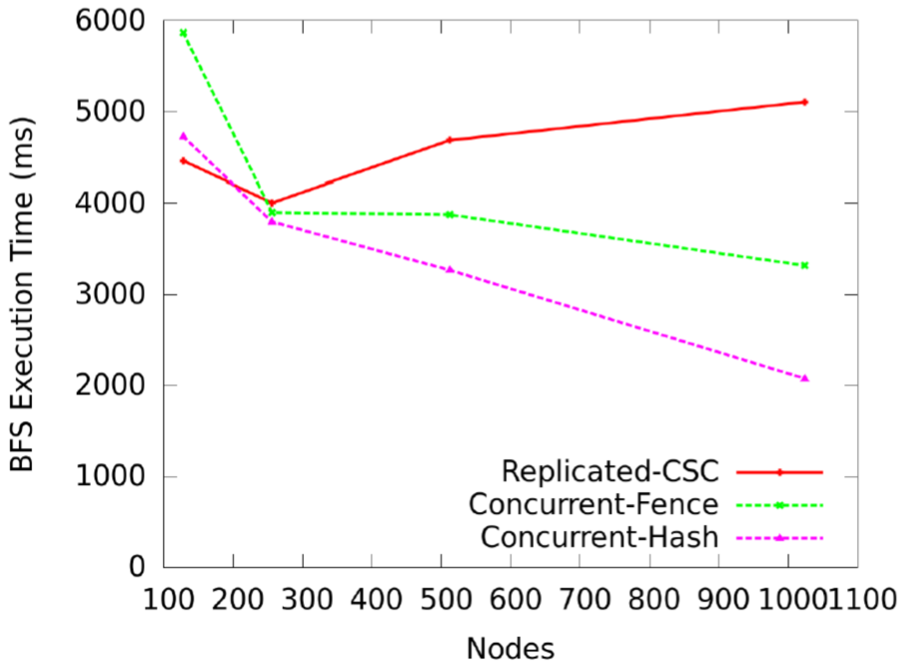
- Cray XC30 w/ 2 x 12-core Intel Xeons
- Cray Aries Interconnect
- Cray MPICH/SHMEM 7.6.0 for the flat tests
- Sandia OpenSHMEM on top of libfabric's GNI provider for hybrid tests

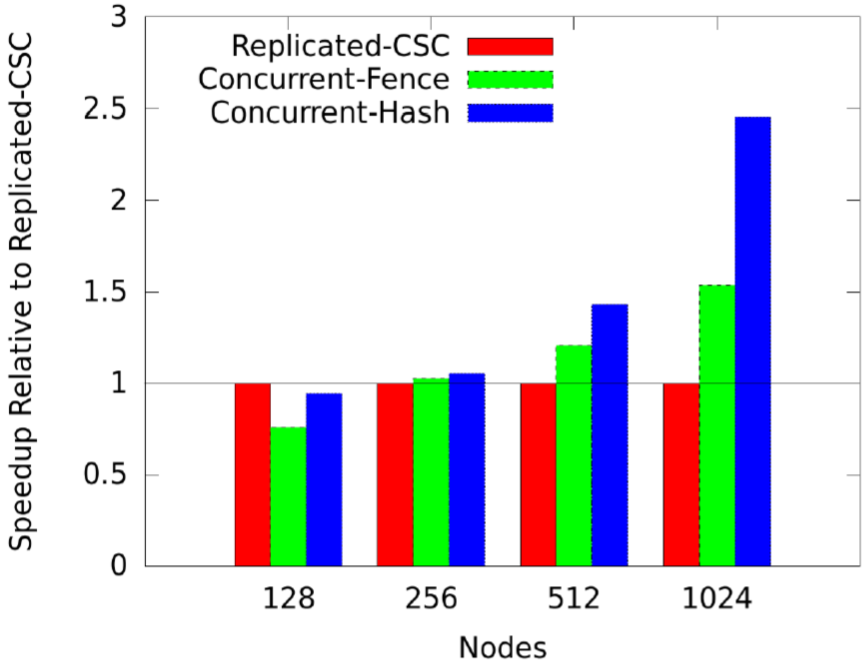
Use graphs with 2^{32} vertices for flat tests, 2^{29} vertices for hybrid.

Comparison to available open source implementations.

Performance Evaluation

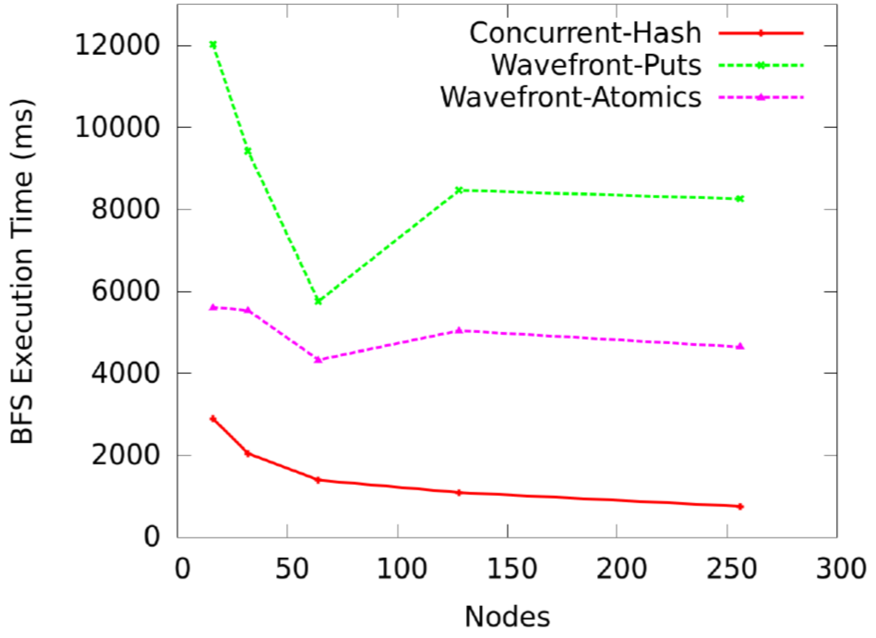
Strong scaling, flat parallelism.

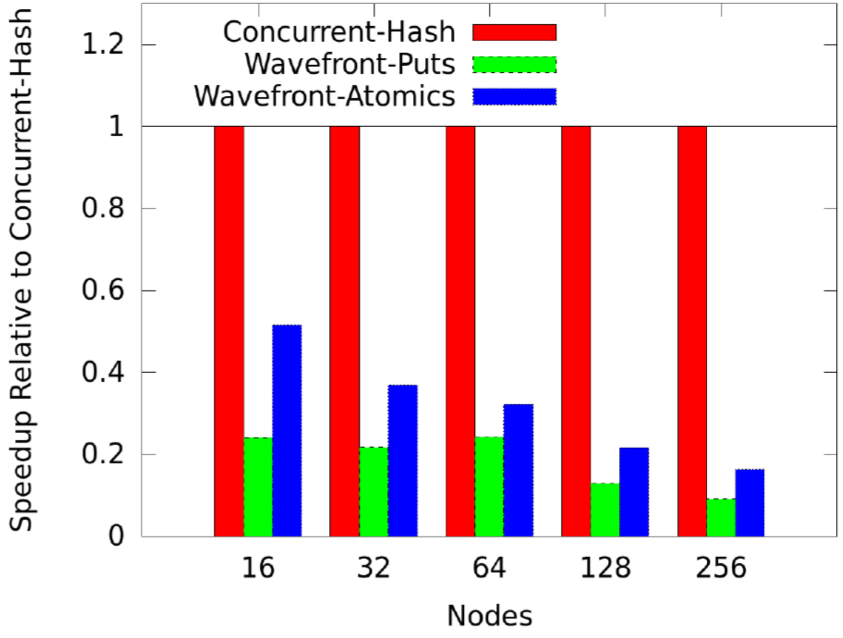




Performance Evaluation

Strong scaling, hybrid parallelism w/ OpenSHMEM contexts.





Outline

Background:
- The OpenSHMEM Programming Model
- Graph500 and the BFS Kernel
- Existing Implementations

Methods:
- Flat OpenSHMEM Implementations
- Hybrid OpenSHMEM Implementations w/ OpenSHMEM Contexts

Evaluation
- Comparison to Reference MPI Implementations

Conclusions

Conclusions

Contributed a survey of the status of currently available, open source OpenSHMEM and MPI Graph500 implementations.

Exploration of techniques in implementing Graph500's BFS kernel on top of OpenSHMEM, using both flat parallelism and hybrid parallelism supported by OpenSHMEM Contexts.

Performance comparison of publicly available OpenSHMEM and MPI implementations on Edison.

Acknowledgements

Los Alamos
NATIONAL LABORATORY
EST.1943

DEPARTMENT OF DEFENSE
UNITED STATES OF AMERICA

RICE

Georgia
Tech

Stony Brook
University

UNIVERSITY of
HOUSTON