Enhanced Deferment for Aggregation Contexts in OpenSHMEM

Aaron Welch, Oscar Hernandez, Stephen Poole

Presented by: Aaron Welch (dawelch@uh.edu)

The Problem

- OpenSHMEM provides a simple PGAS interface for low latency SPMD/RDMA communication that can be efficiently mapped to the hardware
- Some common access patterns can easily congest modern networks with an excessive amount of small messages when implemented with it
- Substantial rewrites of existing code is undesirable, costly, and carries a cognitive burden
- An extension for “aggregation contexts” was introduced to address this issue by deferring messages to send in bulk later for up to 65x improvement, but this alone is insufficient for all cases
- We need to address dependencies between communication operations and use of their results within application code

Bale and Conveyors

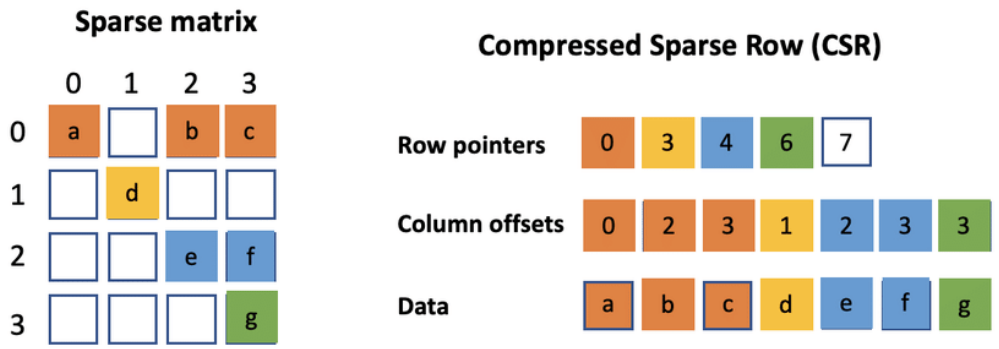
- The bale effort[1] exists to identify key challenges and demonstrate methods that can be used to address them
- Bale provides an aggregation library called conveyors along with a series of applications exhibiting key irregular access patterns implemented either with them or traditional OpenSHMEM atomics, gets, and puts (AGP)
  - Histogram
  - Indexgather
  - Sparse Matrix Transpose
  - Triangle Counting
  - Toposort
  - Etc
- Conveyors are like stateful message queues with operations to `push` and `pull` messages to/from them
  - Implicitly employ two-sided semantics

[1] https://github.com/jdevinney/bale

OpenSHMEM Aggregation Contexts Extension

- Expands on communication contexts from the 1.4 specification (intended for thread isolation)
  - Used to modify existing communication operations to execute with respect to a given context
- Add additional option `SHMEM_CTX_AGGREGATE` for context creation to request conveyor aggregation
- Operations executed on such contexts gain relaxed completion semantics
- Remote completion is only guaranteed upon calling quiet for the context (input buffers reusable)
- Internally, information about an operation of a given size is packed into fixed size structures
- Progress is internally managed

Sparse Matrix Transpose



- Computes the transpose of a sparse matrix in compressed sparse row (CSR) format (seen above [1])
- Difficult to aggregate due to synchronisation as PEs need when determing locations for writing data
- First phase uses a histogram to determine the number of nonzeroes in each column
  - Need new row offsets for storing the results of the transpose $A^{\intercal}$
- Second phase acquires the new location $A^{\intercal}_{j,i}$ for each nonzero $A_{i,j}$ and writes it to the destination PE

[1] Two Sparsities Are Better Than One: Unlocking the Performance Benefits of Sparse-Sparse Networks - Scientific Figure on ResearchGate. Available from: https://www.researchgate.net/figure/The-Compressed-Sparse-Row-CSR-format-for-representing-sparse-matrices-provides-a_fig1_357418189 [accessed 6 Dec, 2023]

Sparse Matrix Transpose Phase 1


```
for (int64_t i = 0; i < A->lnnz; i++) {
    shmem_atomic_inc(&shtmp[A->lnonzero[i]
            / npes], A->lnonzero[i] %
            npes);
}
shmem_barrier_all();
```

OpenSHMEM AGP

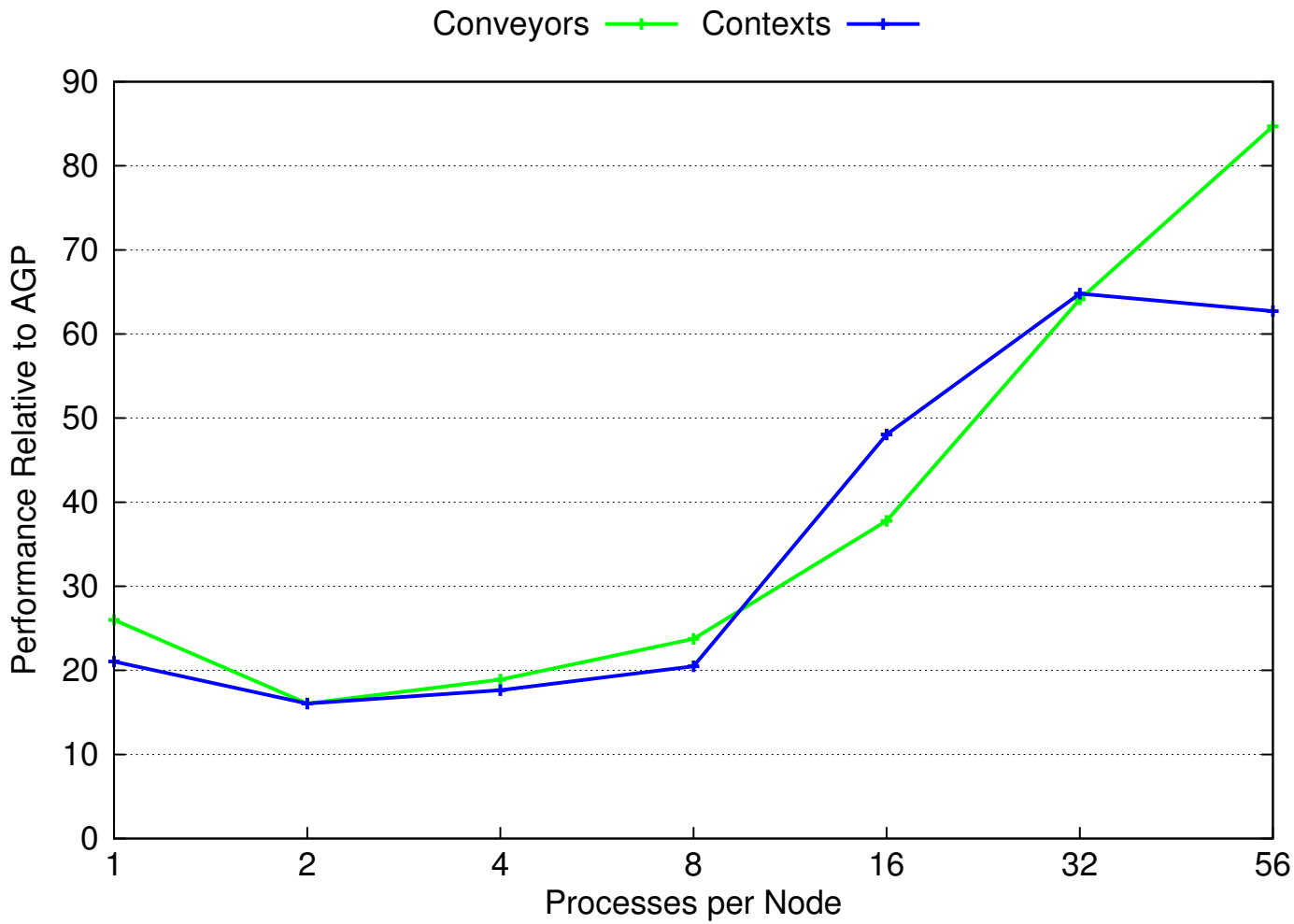
->

```
for (int64_t i = 0; i < A->lnnz; i++) {
    shmem_ctx_int64_atomic_inc(ctx,
            &shtmp[A->lnonzero[i] /
            npes], A->lnonzero[i] %
            npes);
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
```

Aggregation Contexts

- Histogram pattern to get the column counts of `A`

Histogram Performance



- Run with weak scaling on all 32 nodes of the HPC Advisory Council's Iris system using OpenSHMEM from OpenMPI 4.1.3 and NVIDIA ConnectX-6 HDR100 100 Gbit/s InfiniBand adapters

Sparse Matrix Transpose Phase 2


```
//redistribute the nonzeros
for (int64_t row = 0; row < A->lnumrows; row++) {
    for (int64_t j = A->loffset[row]; j <
              A->loffset[row + 1]; j++) {
        int64_t pos =
                  shmem_atomic_fetch_add(&shtmp[A->lnonzero[j]
                  / npes], npes, A->lnonzero[j] %
                  npes);
        shmem_int64_p(&(*At)->nonzero[pos / npes], row
                  * npes + me, pos % npes);
        if (A->value != NULL) {
            shmem_double_p(&(*At)->value[pos / npes],
                      A->lvalue[j], pos % npes);
        }
    }
}
shmem_barrier_all();
```

OpenSHMEM AGP

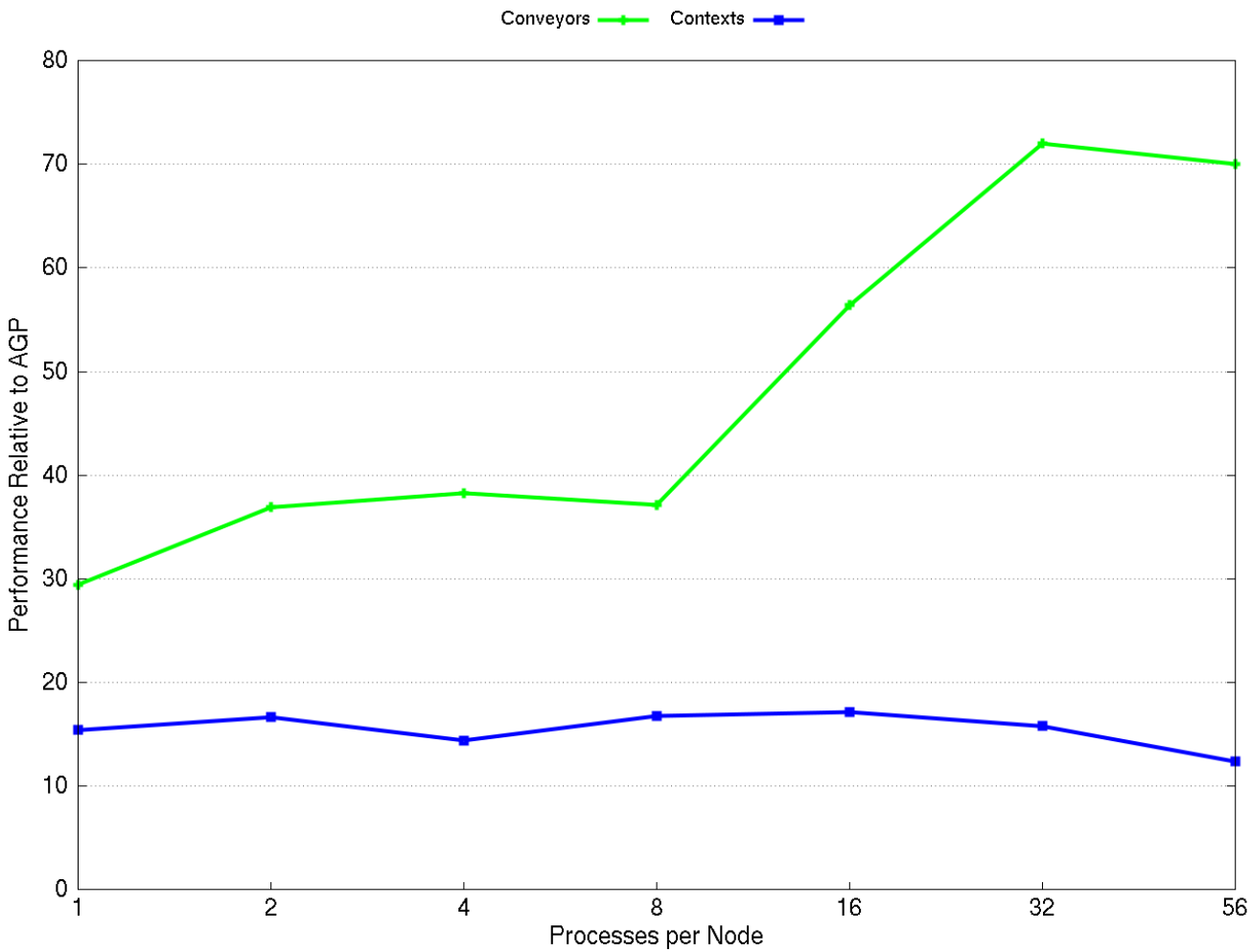
->

```
//redistribute the nonzeros
int64_t *pos[A->lnumrows];
for (int64_t row = 0; row < A->lnumrows; row++) {
    pos[row] = (int64_t *)malloc((A->loffset[row + 1] -
              A->loffset[row]) * sizeof(int64_t));
    if (pos[row] == NULL) {
        FAIL();
    }
    for (int64_t j = 0, col = A->loffset[row]; col + j
              < A->loffset[row + 1]; j++) {
        shmem_ctx_int64_atomic_fetch_add_nbi(ctx,
                  &pos[row][j], &shtmp[A->lnonzero[col
                  + j] / npes], npes, A->lnonzero[col
                  + j] % npes);
    }
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
for (int64_t row = 0; row < A->lnumrows; row++) {
    for (int64_t j = 0, col = A->loffset[row]; col + j
              < A->loffset[row + 1]; j++) {
        shmem_ctx_int64_p_nbi(ctx,
                  &(*At)->nonzero[pos[row][j] / npes],
                  row * npes + me, pos[row][j] % npes);
        if (A->value != NULL) {
            shmem_ctx_double_p(ctx,
                      &(*At)->value[pos[row][j] /
                      npes], A->lvalue[col + j],
                      pos[row][j] % npes);
        }
    }
    free(pos[row]);
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
```

Aggregation Contexts

- Context implementation splits the second phase into two similar to loop fission, fetching all the locations into a buffer then later using those for the writes

Sparse Matrix Transpose Performance



- Run with weak scaling on all 32 nodes of the HPC Advisory Council's Iris system using OpenSHMEM from OpenMPI 4.1.3 and NVIDIA ConnectX-6 HDR100 100 Gbit/s InfiniBand adapters

Now We See the Problem...

- We had to split the second phase due to dependent operations - it first had to fetch a value before using it in a subsequent operation
  - What if a fetched value is used for or within an entire inner loop?
  - What if there are multiple nested levels of that?
- We want to be able to recognise when a symmetric data object associated with a pending fetch is used, and also defer the *capture* of its value
  - What if an application uses a fetched value for more than just OpenSHMEM calls?
- We want to be able to defer *code* until dependent values are fetched as well

What to Do?

- We want to track outstanding get/fetch operations and further defer evaluation of subsequent operations when we detect their results are reused
- Store an index into a table containing additional metadata in the local destination buffer
  - Metadata includes a reference back to the local buffer's address
  - If a symmetric data object is used whose value contains a valid index, must check that the address referenced is the same to avoid coincidental values
- Upon use of a pending operation's result in a future OpenSHMEM call, capture operation type and its arguments to store in a queue for later processing
  - During progress, execute and remove entries whose dependent responses have been processed
  - If the pending queue reaches a maximum size, force progress until operations are released before adding more
- When the response for the operation is processed, clear the metadata and store the final result

Now We Can Do This...


```
for (int i = 0; i < end; i++) {
    shmem_int64_get_nbi(&local[i], &remote[i], 1, pe);
    shmem_int64_put_nbi(&remote[i], &local[i], 1, another_pe);
}
```

Deferred Capture

. . . But We Can't Do This


```
for (int i = 0; i < end; i++) {
    shmem_int64_get_nbi(&local[i], &remote[i], 1, pe);
    shmem_int64_put_nbi(&remote[i], &another_array[local[i]], 1, another_pe);
}
```

Deferred Capture

Deferring Non-OpenSHMEM Code

- We want to be able to have lazy execution that can defer general code which uses pending operation results, similar to tasks
  - Directly accessing/modifying the value, using it as an index, etc
- Unlike before when focusing only on the OpenSHMEM API, we may have any number of dependent operations
- When do we capture non-dependent values (i.e., other variables used in a code block that aren't coming from a fetch like loop indices)?
  - On "task" creation
  - On "task" execution

Deferring Non-OpenSHMEM Code

- We note OpenSHMEM's use of `oshcc` for compilation, similar to MPI and its `mpicc`
- Introduce a small pragma API for deferring statements or blocks (`#pragma shmem defer`)
- Dependent fetch results can be explicitly specified (i.e., `defer(myvalue)`) or automatically inferred by the compiler
- Capture fetch results on execution by default, or optionally immediately upon the associated operation's completion if requested with an `immediate(myvalue)` clause
- Capture stack variables on "task" creation by default, or optionally on execution with a `shared(foo)` clause
- Similar to before, deferred code blocks are put in pending queues, except that their execution and release from these queues requires completion of *all* dependent fetches

Back to Sparse Matrix Transpose


```
//redistribute the nonzeros
for (int64_t row = 0; row < A->lnumrows; row++) {
    for (int64_t j = A->loffset[row]; j <
            A->loffset[row + 1]; j++) {
        int64_t pos =
                shmem_atomic_fetch_add(&shtmp[A->lnonzero[j]
                / npes], npes, A->lnonzero[j] %
                npes);
        shmem_int64_p(&(*At)->nonzero[pos / npes], row
                * npes + me, pos % npes);
        if (A->value != NULL) {
            shmem_double_p(&(*At)->value[pos / npes],
                    A->lvalue[j], pos % npes);
        }
    }
}
shmem_barrier_all();
```

OpenSHMEM AGP

->

```
//redistribute the nonzeros
for (int64_t row = 0; row < A->lnumrows; row++) {
    for (int64_t j = A->loffset[row]; j <
            A->loffset[row + 1]; j++) {
        static int64_t *pos;
        shmem_ctx_int64_atomic_fetch_add_nbi(ctx, pos,
                &shtmp[A->lnonzero[j] / npes], npes,
                A->lnonzero[j] % npes);
        #pragma shmem defer immediate(pos)
        {
            shmem_ctx_int64_p_nbi(ctx,
                    &(*At)->nonzero[pos / npes], row
                    * npes + me, pos % npes);
            if (A->value != NULL) {
                shmem_ctx_double_p(ctx,
                        &(*At)->value[pos / npes],
                        A->lvalue[j], pos % npes);
            }
        }
    }
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
```

Aggregation Contexts

Status and Future Work

- Design looks promising, but is still undergoing revisions
- Complete implementation of `defer` pragma planned for LLVM and eventually GCC
- Current testing being performed on ORNL's Frontier
- We intend to use these enhancements to improve Bale's transpose and triangles applications, with the goal of reaching a completed toposort

Related Work: OpenSHMEM Queues

- OpenSHMEM queues [1] are another proposed method of aggregation
- Introduces a new API for explicitly pushing and popping messages onto queues
  - Allows for QoS support for message prioritisation
  - Requires more substantial changes to application code
- Operation is more transparent
- No way to support deferred execution of code blocks

[1] Vishwanath Venkatesan and Manjunath Gorentla Venkata. 2023. OpenSHMEM Queues: An abstraction for enhancing message rate, bandwidth utilization, and reducing tail latency in OpenSHMEM Applications. In Workshops of The International Conference on High Performance Computing, Network, Storage, and Analysis (SC-W 2023), November 12–17, 2023, Denver, CO, USA. ACM, New York, NY, USA 10 Pages.

Thank You

You can reach me for further questions at dawelch@uh.edu

Previously published results for aggregation contexts:

Welch, A., Hernandez, O., and Poole, S. Extending OpenSHMEM with Aggregation Support for Improved Message Rate Performance. In Euro-Par 2023: Programming, Compilers, and Performance: 29th International Conference on Parallel and Distributed Computing, Limassol, Cyprus, August 28-September 1, 2023.