Enhancing Small Message Aggregation with Directive-Based Deferred Execution

Aaron Welch*, Oscar Hernandez*, Stephen Poole†, and Wendy Poole†
*Oak Ridge National Laboratory, Oak Ridge, Tennessee, USA
Email: welchda@ornl.gov, oscar@ornl.gov
†Los Alamos National Laboratory, Los Alamos, New Mexico, USA
Email: swpoole@lanl.gov, wkpoole@lanl.gov

***Abstract*—The partitioned global address space (PGAS) model offers one-sided communication operations to efficiently access local and remote data through a distributed shared memory model using point-to-point network operations. An extension to the OpenSHMEM PGAS library previously demonstrated how message aggregation could be applied in a minimally intrusive manner to an application, while still achieving a significant portion of the performance possible through manual tuning. However, its primary deficiency was the inability to abstract dependencies between aggregated remote memory accesses and their subsequent uses, which must be managed explicitly by applications. This undermined its goal of preserving algorithmic intent. In this paper, we present a novel directive-based approach for automatically deferring the execution of arbitrary code that depends on aggregated messages, shifting the concern of their efficient management from the application to the implementation. We demonstrate our approach using two applications from the bale 3.0 classic suite on the Frontier supercomputer.**

***Index Terms*—OpenSHMEM, Message Aggregation, Conveyors, Deferred Execution, Compilers**

I. Introduction

As HPC systems have evolved, particularly in response to the end of Moore's Law, the design of network interconnects has shifted [1]. Traditionally, the focus was on optimising for high-bandwidth and bulk data transfers, often at the expense of latency and message rate. However, with the emergence of new workloads in machine learning and data analytics, there is an increasing need to address not only bandwidth *density* but also improve message rates and reduce latencies.

This comes partially as a result of applications dependent on high rates of small messages, a pattern frequently seen in data analytics applications using many-to-many communication and irregular access patterns. Recent network interconnect technologies, like those in Slingshot 11, have made improvements in this area [2] through the use of congestion control, dynamic routing, and quality of service in their switches. However, the gap between applications well tuned for high-bandwidth use and those suffering from the small message problem remains vast. While PGAS programming models offer a distributed shared memory abstraction that is well-suited for developing algorithms with irregular access patterns, optimising network throughput generally requires the use of explicit message aggregation strategies. These involve extensive code transformations in point-to-point algorithms where aggregation cannot be abstracted through the use of network collectives, which is costly to develop and maintain. This additional complexity can significantly diminish the productivity benefits that a PGAS model is intended to provide.

Prior work on the OpenSHMEM PGAS library proposed an extension called "aggregation contexts" based on an aggregation library called conveyors and aimed at addressing this issue by expanding the API for non-blocking communication operations. It does so by conditionally changing their completion semantics such that data for many remote operations can be accumulated and later sent in bulk to their respective targets for local processing [3]. While this approach worked well for independent operations and successfully scaled up to 8,192 nodes [4], its usefulness was limited when there were dependencies among the data that required aggregation. That is, if some code were to fetch remote data and then use it in a subsequent operation (e.g., as an index into another remote array), it would not be a candidate for context aggregation.

Our solution is to employ user-provided hints that annotate regions of code to be deferred via a directive-based API. This allows the compiler to transform dependent code, deferring its execution based on the tracked status of the outstanding aggregated communication it's dependent upon. Our design requires no code modifications beyond a single directive for each lexical scope containing communication operations for which the user wants to defer dependent code.

This paper is organised as follows. An overview of related work can be seen in Section II. Section III provides some background on conveyors and aggregation contexts, followed by a description of how contexts were enhanced to support deferred execution in Section IV. In Section V, we demonstrate the performance improvements of our approach using the sparse matrix transpose and triangle counting bale applications on the Frontier supercomputer at ORNL. Finally, concluding remarks and directions for future work are discussed in Section VI.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00094

II. Related Work

The work in [5] also introduces conveyor-style aggregation to the PGAS paradigm by proposing an extension to the task-parallel Habanero-C Library (HCLib) using the actor model. In this framework, actors can possess multiple *mailboxes*, allowing them to send and receive fine-grained active messages backed by conveyor aggregation. Since the abstraction around the conveyor API is thin, it requires developers to explicitly manage dependencies between aggregated data accesses as well as the state and progression of conveyors, including handling failure conditions of push/pull operations due to buffer limitations. This approach, while offering powerful low-level control, requires a detailed understanding of the underlying communication patterns and explicit handling of dependencies, reducing its abstraction level.

Grossman et al. [6] explored the integration of asynchronous task parallelism within OpenSHMEM by extending HCLib to support task-based programming models. Their work introduced an abstraction layer that enables fine-grained task parallelism, but it requires explicit handling of dependencies between tasks, making the model powerful but less abstract. This approach is particularly useful for applications that need to manage complex dependencies, although it demands a deep understanding of the underlying communication patterns.

Lu, Curtis, and Chapman [7] proposed an extension to OpenSHMEM that introduces active message support for task-based programming. Their approach leverages the existing OpenSHMEM infrastructure to support active messages, allowing for asynchronous communication. The explicit nature of handling dependencies between active messages in this model would require developers to carefully manage the state of communication, similar to other low-level approaches.

Other directive-based programming models like OpenMP provide directives to express dependent tasks, but they don't focus on dependent remote or local memory access or network operations. However, the new detach clause introduced in OpenMP 5.0 is meant to interoperate with asynchronous libraries, but does not offer a way to abstract small message aggregation.

III. Background

First, we will provide a brief background on conveyors, as our past and present work is based upon it. We will then provide an overview of aggregation contexts as they were previously designed so as to provide a foundational basis for understanding our changes for implementing deferred execution.

A. Conveyors

Conveyors [8] provide an abstraction around message queues with which fixed-size items can be *pushed* or *pulled*. The size and contents of these buffers are specified and processed solely by the user — the responsibility of conveyors lies with packing many of them into incoming/outgoing buffers for bulk transmission to target processes.

Use of conveyors implicitly requires the creation of ad-hoc progress loops each time they are needed, which continuously cycle between filling outgoing queues and processing items from incoming ones until all communication requests are complete. Messages are physically sent across the network either as relevant buffers fill or during explicit calls to *advance* them, though completion of any specific operation is not guaranteed until an advance indicates that the current communication epoch is complete.

B. Aggregation Contexts

The proposed aggregation contexts extension for OpenSHMEM [3] is implemented on top of conveyors to abstract their usage on applications. They were designed with the goal of minimally changing existing code and thus preserving its algorithmic intent, even if it comes at the sacrifice of some additional performance that could have been gained through direct conveyor use. To achieve this, they introduced the ability to create a logical context object that indicates that aggregation is desired, which must be passed as an additional argument to all the basic atomic, get, and put (AGP) operations. This implicitly changes the completion semantics of each such operation such that neither local nor remote completion is guaranteed until the user executes a *quiet* on the context, signaling the start of a new communication epoch in which the cycle continues until the user eventually destroys the context.

These contexts are able to map a diverse set of communication operations to a single set of conveyors by creating a fixed-size unified packet structure capable of describing any possible communication request supported by OpenSHMEM, generally consisting of a type identifier, local and remote addresses, and a value. Due to the fixed-size nature of these packets that conveyors mandate, this necessarily results in drops in the efficiency of network utilisation for operations not requiring values for all members of the packet encoding, proportional to the wasted encoding space. Individual calls to AGP operations essentially just translate to encoding the relevant information into one of these packets and pushing it onto a conveyor, where the majority of the work including the actual execution of operations is left to the context's internal progress management, which follows a very similar pattern to what was described for conveyors' ad-hoc progress loops with the addition of type checks to determine how to respond to pulled requests.

This approach worked very well, as long as all communication within an epoch was able to be executed independently – that is, when no result of an operation would be accessed until after the next quiet. However, when such dependencies were introduced, they required more effort from the application developer to restructure their code to accommodate aggregation (e.g., by prefetching a large number or all such results and performing a quiet before proceeding to use them in the subsequent code). An example of this can be seen in the bale application for sparse matrix transpose. Its first phase performs histogram calculations without dependent operations to obtain column counts. However, its second phase repeatedly fetches

positions in the resulting transpose matrix before subsequently writing to them, as shown in Listing 1.

```
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();
```

Listing 1: Transpose Phase 2 Using AGP

Using the aggregation contexts extension alone requires developers to manually perform loop fission, similar to what a compiler might do, with a quiet operation added between the two smaller loops, as shown in Listing 2. This approach not only places a greater burden on the development and maintenance of applications, but the impact can be even more pronounced if multiple dependencies exist within an epoch, particularly when nested within loops. This detracts from aggregation contexts' goal of minimising required code restructuring and is what we seek to avoid with our approach for supporting deferred execution.

```
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_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_int64_p_nbi(ctx,
            &(*At)->nonzero[pos[row][j] / npes], row *
            npes + me, pos[row][j] % npes);
        if (A->value != NULL)
            shmem_double_p_nbi(&(*At)->value[pos[row][j]
                / npes], A->lvalue[col + j],
                pos[row][j] % npes);
    }
    free(pos[row]);
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
```

Listing 2: Transpose Phase 2 Using Aggregation Contexts

IV. Deferred Execution

Our solution was to provide a directive-based abstraction that tracks the status of any outstanding aggregated operations and automatically defers the execution of code dependent upon their results. Applications would simply annotate blocks containing remote fetching operations whose dependent code they want deferred using `#pragma shmem defer`. Listing 3 shows how the defer directive is applied to the second phase of the sparse matrix transpose code.

```
for (int64_t row = 0; row < A->lnumrows; row++) {
#pragma shmem defer
    for (int64_t j = A->loffset[row]; j <
         A->loffset[row + 1]; j++) {
        int64_t pos = shmem_atomic_fetch_add(ctx,
            &shtmp[A->lnonzero[j] / npes], npes,
            A->lnonzero[j] % npes);
        shmem_int64_p_nbi(ctx, &(*At)->nonzero[pos /
            npes], row * npes + me, pos % npes);
        if (A->value != NULL)
            shmem_double_p_nbi(ctx, &(*At)->value[pos /
                npes], A->lvalue[j], pos % npes);
    }
}
shmem_ctx_quiet(ctx);
shmem_barrier_all();
```

Listing 3: Transpose Phase 2 with Deferred Execution

Its implementation involves a compiler plugin that must scan annotated blocks for regions that are dependent upon prior fetch requests, extract them into callback functions, and store relevant stack data along with a counter set to the number of associated fetch operations into a defer queue. As fetch operations get completed, their associated counter gets decremented until it eventually reaches zero, at which point its dependent block is ready to be executed. If the defer queue reaches a set maximum size, the context must force and wait on progress of outstanding AGP operations until it can clear more space by executing blocks previously waiting in the queue. Execution order of deferred blocks is strictly first in, first out.

One of the benefits of implementing deferred execution capabilities in the compiler as we are proposing here is that it can open up the door to all manner of new optimisation capabilities. By adding semantic knowledge of the OpenSHMEM API to compiler analyses, it allows for transformations that may not have been possible before, such as reordering of communication operations or other statements in relation to them, or other optimisations like code hoisting. This could not only have great impact in relation to deferred code blocks, but could potentially improve ordinary OpenSHMEM related code that is neither aggregated nor deferred.

V. Evaluation

We evaluated our deferred implementation based on two of the bale 3.0 classic applications: sparse matrix transpose and triangle counting. Transpose is a straightforward use case as seen in Section IV, while triangle counting expands this to a more complicated graph algorithm with multiple levels of nested deferment. As we have yet to implement our design in its entirety into the compiler, what follows is an initial proof of concept where some of the necessary transformations were applied by hand.

The following runs were performed on 256 nodes of ORNL's Frontier, using 1–64 processes per node with a block distribution. Each Frontier compute node has a single 64-core AMD Epyc 7A53 2 GHz CPU with 512 GB of memory and four HPE Slingshot 11 200 Gbit/s NICs connected through four AMD MI250X GPUs. We compiled with Cray clang 15.0.0 and ran using Cray OpenSHMEM-X 11.7.2.3 with XPMEM using the SLURM hint "–hint=nomultithread" for optimal core bindings. Our results obtained using aggregation contexts are compared against bale's versions of the applications using direct, hand-tailored conveyor code in terms of speedup relative to the original AGP versions (i.e., a result of 10 would indicate 10 times faster execution compared to AGP).

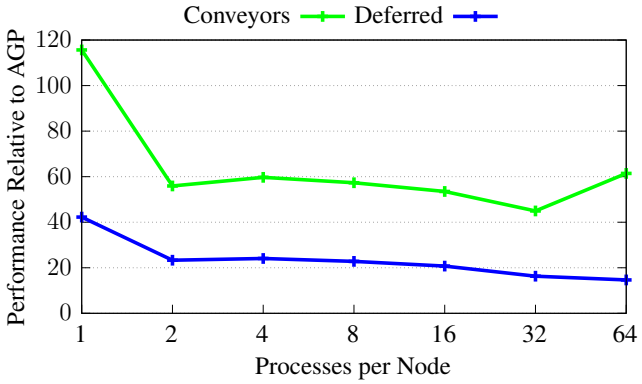

Fig. 1: Matrix Transpose Speedup over AGP

The results for transpose can be seen in Figure 1. Aggregation contexts performed quite favourably here, generally achieving a consistent fraction of conveyors' speedup, which ranged between 2–3 times faster. Similarly, the results for triangle counting can be seen in Figure 2. Aggregation contexts took a slightly larger performance hit here, for which the relatively larger overhead for the nested deferment increased conveyors' relative perforamance in excess of 4x.

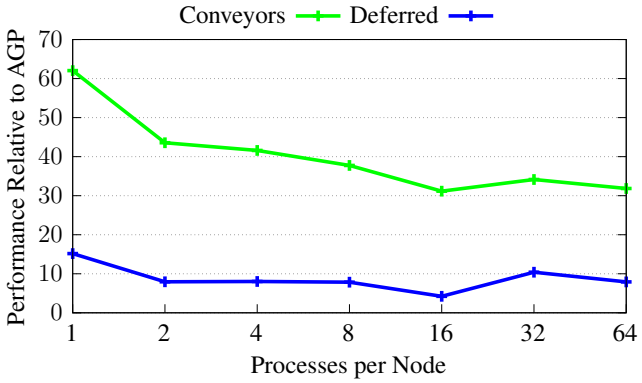

Fig. 2: Triangle Counting Speedup over AGP

As according to their design, the overhead of using aggregation contexts on application developers and their code is extremely minimal, and instead all the opportunity costs are tied to their runtime performance compared to a manually tailored result. Two of the main sources for such runtime overhead are the encoding efficiency of the buffer items supplied to the conveyor (which as suggested in [4] could still be a good focus for further optimisation, the gains for which we estimate could be up to 50%), and that of managing the storage and retrieval of execution state for deferred blocks (which may also have room for additional improvement). However, even with our initial implementation, the benefits are clearly massive and we think can already outweigh the cost of manually implementing the alternative in user applications. There are most likely also a number of optimisation opportunities, though such considerations were left for future investigation.

VI. Conclusion

In this paper, we introduced a directive-based approach to abstract small message aggregation capabilities of dependent operations in PGAS programming models through OpenSHMEM. Our method leverages compilers to implement automatic deferred execution that abstracts dependencies between memory accesses, enabling more efficient aggregation without compromising the application's algorithmic intent or requiring significant code restructuring. The results from our evaluation show that our approach effectively removes blocking operation and synchronisation overheads in the sparse matrix transpose and triangle counting applications from the bale 3.0 classic suite, resulting in substantial performance gains of up to 10–20x improvements to application runtime over non-aggregated execution. Beyond fully implementing our design within the LLVM framework, other avenues for future work include a focus on refining the directive interface, testing it on a wider array of applications, and investigating optimisation opportunities and the compiler analyses that might be needed to support them. It could also be worth looking into adapting our strategies to other PGAS environments, or exploiting SmartNICs to offload handling of deferred blocks to them for potential further performance gains.

VII. Acknowledgements

This research used the Frontier and Andes resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This work was funded through Strategic Partnership Projects Funding Office via Los Alamos National Laboratory with IAN 619215901.

References

[1] J. Shalf, "Hpc interconnects at the end of moore's law," in *Optical Fiber Communication Conference (OFC) 2019*. Optica Publishing Group, 2019, p. Th3A.1. [Online]. Available: https://opg.optica.org/abstract.cfm?URI=OFC-2019-Th3A.1

[2] D. De Sensi, S. Di Girolamo, K. H. McMahon, D. Roweth, and T. Hoefler, "An in-depth analysis of the slingshot interconnect," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2020, pp. 1–14.

[3] A. Welch, O. Hernandez, and S. Poole, "Extending openshmem with aggregation support for improved message rate performance," in *Euro-Par 2023: Parallel Processing*, J. Cano, M. D. Dikaiakos, G. A. Papadopoulos, M. Pericàs, and R. Sakellariou, Eds. Cham: Springer Nature Switzerland, 2023, pp. 32–46.

[4] A. Welch, O. Hernandez, W. Poole, and S. Poole, "Scalable small message aggregation on modern interconnects," in *VIVEKfest 2024: In Honor of Vivek Sarkar's Contributions to Parallelism and Programming Languages, part of SPLASH 2024*, 2024, to appear. [Online]. Available: https://2024.splashcon.org/home/vivekfest-2024

[5] S. R. Paul, A. Hayashi, K. Chen, and V. Sarkar, "A scalable actor-based programming system for PGAS runtimes," *CoRR*, vol. abs/2107.05516, 2021. [Online]. Available: https://arxiv.org/abs/2107.05516

[6] M. Grossman, V. Kumar, Z. Budimlić, and V. Sarkar, "Integrating asynchronous task parallelism with openshmem," in *OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments*, M. Gorentla Venkata, N. Imam, S. Pophale, and T. M. Mintz, Eds. Cham: Springer International Publishing, 2016, pp. 3–17.

[7] W. Lu, T. Curtis, and B. Chapman, "Openshmem active message extension for task-based programming," in *OpenSHMEM and Related Technologies. OpenSHMEM in the Era of Exascale and Smart Networks: 8th Workshop on OpenSHMEM and Related Technologies, OpenSHMEM 2021, Virtual Event, September 14–16, 2021, Revised Selected Papers*. Berlin, Heidelberg: Springer-Verlag, 2021, p. 129–143. [Online]. Available: https://doi.org/10.1007/978-3-031-04888-3_8

[8] F. M. Maley and J. G. DeVinney, "Conveyors for streaming many-to-many communication," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 1–8.