OpenSHMEM Active Message Extension for Task-Based Programming

Wenbin Lu, Tony Curtis, and Barbara Chapman

Institute for Advanced Computational Science
Stony Brook University
Stony Brook, USA
{wenbin.lu, anthony.curtis, barbara.chapman}@stonybrook.edu

Abstract. As a lightweight library-based Partitioned Global Address Space (PGAS) programming model, OpenSHMEM provides efficient one-sided and collective communications and is receiving more attention in recent years. However, task-based programming models are getting bigger traction in scientific computing communities. Application developers are attracted by their ability to achieve better load balance in the face of ever-growing application complexity, and the increasing on-node parallelism in modern high-performance computing machines. Although communication contexts provide threads with first-class access to the network in the OpenSHMEM+X model, OpenSHMEM still has very limited ability to perform advanced operations found in other task-based models. For example, compared to the remote procedure call (RPC) mechanism in the UPC++ programming model, more work is required if the signal/wait routines are used to achieve similar remote task launching operations. In this paper, we introduce a lightweight active message (AM) extension to OpenSHMEM that is designed to perform short, non-blocking remote function invocations. This extension aims to bring some benefits of task-based programming to OpenSHMEM without making it a full-blown heavyweight tasking system with a sophisticated scheduler. We study the performance of this active message extension by running micro-benchmarks, and by evaluating its computation efficiency at different task granularities using the TaskBench framework.

Keywords: PGAS, OpenSHMEM, Active Message, Tasking

1 Introduction

The increasing complexity of software and hardware in the exascale era calls for more adaptive and dynamic programming models. In recent years we are seeing a surge of new programming models that break away from the traditional SPMD-way of expressing parallelism and embrace the flexibility and scalability provided by dynamic execution of operations, mostly in the form of tasks or a variant of active messages [8]. In today's diverse and heterogeneous HPC systems, an application using HPX [11], an active global address space programming model, was able to achieve a much higher parallel efficiency at scale than MPI [6].

Many members in the family of partitioned global address space (PGAS) programming models already have some way to launch functions remotely: Coarray Fortran [20], GASNetEX [3]/UPC++ [1], and Chapel [4] are among the most prominent ones. Their ability to insert operations into another process's execution flow is a simpler, sometimes more scalable, way to realize tasks and their dependencies than using distributed signal variables. As of OpenSHMEM 1.5 [15], a similar feature is not available, and we believe it is affecting OpenSHMEM's adoption in the supercomputing community.

In this paper, we propose a lightweight active message extension for OpenSHMEM that is designed to support a basic form of task-based programming without feeling out-of-place when mixed with other parts of the model. It enables efficient remote invocation of pre-defined message handler functions across Processing Elements (PEs). Our work provides a basis for the inclusion of this important feature, and we hope to spark discussion in the OpenSHMEM community on this topic.

This paper is organized as follows: Section 2 discusses the background of this work. Section 3 describes the proposed API extension and its implementation. Section 4 provides a preliminary evaluation of the work and Section 5 gives a conclusion and talks about future work.

2 Background

2.1 Task-Based Programming and Active Messages

Task-based programming is the practice of expressing parallelism in terms of small units of computation called tasks. Two tasks can be executed independently or have an order imposed on them in the form of task dependency. The directed acyclic graph (DAG) constructed with task dependencies is expressive enough for the majority of HPC applications, while still can be mapped and executed on supercomputers. Due to its ability to express sophisticated workflows in an organized manner and expose more opportunities for load balancing, task-based programming is playing an increasingly prominent role in the exascale era.

Active messages [8] enable one process to schedule a function (AM handler) invocation on a remote process, using a pre-registered AM handler ID and a set of handler arguments (AM payload) contained in the scheduling request. AM and its derivatives have become building blocks of many HPC programming models and machine learning frameworks for performing distributed control flow (e.g. work assignments, load balancing). While a simple AM implementation lacks the central/distributed task scheduler found in fully-fledged tasking frameworks, it still can be used to construct tasks and dependencies between them. The developer needs to implement an application-specific scheduler to decide when and where to send AM requests, but this might be preferable for some applications.

2.2 OpenSHMEM

OpenSHMEM[5] is an SPMD programming model that implements the PGAS memory model. It is library-based and uses one-sided remote memory access (RMA) as its main method for doing point-to-point communication and synchronizations between its processes/PEs. Due to its elegant and implementer-friendly design, it has been well-received in both academia and commercial products like NVSHMEM [14].

However, when compared to other members of the PGAS family, OpenSHMEM lacks advanced features that could help developers achieve scalability and portability in the face of the exploding complexity of the modern HPC ecosystem. The recent addition of teams [17] and contexts [7] are solid steps towards this direction and have been shown to improve application performance [13], but data movement is still the main focus.

Currently, if the developers want to have a task-like workflow in an OpenSHMEM application, the entire DAG of tasks must be hard-coded into the application, with every task spawning operation and dependency realized through a dedicated signal variable and a wait operation. This approach not only loses the flexibility of task-based programming but also requires heroic effort and is very error-prone. Moreover, inputs and outputs of the tasks must be placed on the symmetric heap and passed using PUTs or GETs even if they are single integers/floating-point numbers, which further increases complexity and reduces scalability.

The active message extension proposed in this paper could bridge the gap between OpenSHMEM and other programming models on handling control flow on the distributed-memory level. Instead of using signals to trigger task execution and notify the availability of execution results of tasks, parent tasks can inject tasks into any PE's execution flow and the children tasks can invoke AM handlers on the parent task's PE to fetch computation results, or simply send the results as the AM payload if the sizes are within the limits. If designed with care, the AM extension will blend nicely with the rest of the OpenSHMEM specification and does not introduce unnecessary overhead to other OpenSHMEM operations.

2.3 Related Work

An AM extension for OpenSHMEM has been proposed before [10], in which a set of APIs is presented to initiate, progress, and perform active message operations between PEs. Their GASNet-based implementation features opportunistic execution of incoming active messages using a background polling thread. The potential risk of data race on internal data structures and other progression issues leads to the need of banning the invocation of most OpenSHMEM routines from the AM handler, as well as a dedicated mutex interface for handler safety. The main difference between their design and ours is that: we allow the use of simple point-to-point communication routines including sending AMs from the handler, give the user total control of where the handler gets executed, and let the user handle thread-safety using whatever mechanism they see fit. As a result,

our design does not require `SHMEM_THREAD_MULTIPLE` when not requested by the application and is much more flexible in what can be done inside the handler.

MPI, being the most popular distributed-memory programming model for HPC, also has had a few attempts to retrofit it with active message capabilities. The MPI-Interoperable Generalized Active Messages [24] is the most complete one, which is similar to the `MPI_Op` of MPI-3's RMA accumulate operation but is extended with user-defined operations and a data streaming-like interface. Compared to our design, their API is extremely complex and requires more effort to use. Another attempt at MPI AM is presented in [21] and has achieved good performance. However, the work focused on implementing an active message mechanism using MPI RMA and did not design a general-purpose API interface for it.

Active message-like functionalities can be found in many other programming models. UPC++'s RPC mechanism supports automatic serialization and provides future objects that can be waited on to obtain the RPC's return value. Charm++ [12] features location-agnostic method invocations on a unified view of all the distributed C++ objects, with its runtime performs automatic object migration and load balancing behind the scene. Legion [2] implements a mapper layer that controls the placement of the tasks instead of having to specify on which process each task should run in the application's main workflow. These advanced features are too heavy-weight for both the OpenSHMEM specification and its implementations.

3 Design and Implementation

This section describes the design of our OpenSHMEM active message extension and the rationale behind it. The extension's implementation is also discussed to demonstrate how we are able to allow the use of simple point-to-point communication operations from within the AM handlers.

3.1 OpenSHMEM Active Message API Extension

The type and macro definitions for the AM handler are shown in Listing 1.1. The size of the active message payload is limited by the implementation-defined macro `SHMEMX_AM_PAYLOAD_MAX_SIZE`, typically this will be a few kilobytes. We decide to add this restriction to avoid introducing the rendezvous protocol to handle large payload sizes. Since OpenSHMEM focuses on fast one-sided communication operations, an active message interface with MPI-like request objects and/or callbacks deviates too far away from OpenSHMEM's communication semantics. If the user needs to transfer a large amount of data with an AM request, our design allows invoking PUTs/GETs from within the AM handler so it should not be a problem.

Active message handlers must be registered on the destination PE before the initiator PE can schedule its execution. The handlers must have the same function type defined by `shmemx_am_handler_t`, and its definition must be visible

```
// Maximum active message payload size.
#define SHMEMX_AM_PAYLOAD_MAX_SIZE

// Active message handler signature.
typedef void (*shmemx_am_handler_t)(void*  payload,
                                    size_t length,
                                    void*  args_r,
                                    void*  args_p,
                                    int    source_pe)
  [IN] payload    Active message payload
  [IN] length     Size of the payload
  [IN] args_r     Registration-time user arguments
  [IN] args_p     Polling-time user arguments
  [IN] source_pe  PE number of the initiator of this AM
```

Listing 1.1. Proposed OpenSHMEM Active Message API type definitions

to the compiler/linker when the application is compiled. When a handler is invoked by the destination PE, the payload and its size, a pointer provided by the user when the handler was registered, a pointer passed to the polling routine that invoked this handler, and the PE number of the source of the AM request are passed as function arguments. Once the handler returns, the payload buffer is freed.

One of the main advantages of our design is the ability to call OpenSHMEM routines from the AM handler. Point-to-point communication operations like PUT/GET/atomic operations are all supported, as well as `shmem_fence` and `shmem_quiet`, and even send AM requests using the API in Listing 1.2. Usage of collective communications, distributed locking routines, symmetric heap management routines, and contexts & teams routines inside the handlers are still forbidden. Additionally, the user can make `libc` and other external function calls inside the AM handler, but we strongly discourage performing any potentially blocking operation in the handler, as doing so could prevent timely handling of other AM requests or even cause deadlocks. Our proposed API is inter-operable with shared-memory tasking frameworks, so long-running computations or system calls like I/O can be handled by offloading them to other threads.

Listing 1.2 defines the API extension that will be used to register, send and progress OpenSHMEM active messages. A PE can call `shmemx_am_set_handler` to register an AM handler and receive an integer as the ID of the registered handler, this ID will be used by an initiator PE to send an AM request that invokes this handler. The registration routine is not a collective call and the application is not required to have the same handler-to-ID mapping across all PEs. A pointer to local arguments can be registered along with the handler, and it will be passed as the third argument (`args_r`) to the handler for every AM request of this ID.

```
// Set & reset active message handler.
void shmemx_am_set_handler(shmemx_am_handler_t handler,
                           void*               args_r,
                           int*                id)
  [IN]  handler   Active message handler (NULL to reset)
  [IN]  args_r    User-defined local arguments
  [OUT] id        Active message ID

// Send an active message.
void shmemx_am_send_nbi(int    id,
                        void*  payload,
                        size_t length,
                        int    pe)
  [IN] id          Active message ID
  [IN] payload     Payload to send
  [IN] length      Size of the payload
  [IN] pe          PE number of the remote PE

// Non-blocking poll of incoming active messages.
int shmemx_am_poll(void* args_p)
  [IN]     args_p  User-defined local arguments
  [RETURN] Non-zero if any AM was processed, zero otherwise.

// Blocking wait on incoming active messages.
void shmemx_am_wait(void* args_p)
  [IN] args_p      User-defined local arguments
```

Listing 1.2. Proposed OpenSHMEM Active Message API routines

To send an AM request, the initiator PE should pass the ID, the payload and its size, and the PE number of the destination to the `shmemx_am_send_nbi` routine. Safe reuse of the payload buffer is not guaranteed when the non-blocking send routine returns and so the user should use the `shmem_quiet` routine to wait for the completion of all outbound AM requests, as the AM requests are sent through the default context `SHMEM_CTX_DEFAULT`. We do not provide any guarantee of the ordering of consecutive AM requests, as well as the atomicity of the execution of the handlers.

For the progression and completion of active message requests, we provide the `shmemx_am_poll` and `shmemx_am_wait` pair of routines. The `shmemx_am_poll` routine checks for arrived AM requests, it returns 0 immediately if no pending requests could be found, or a non-zero number if it was able to execute one or more AM requests. Alternatively, if the application calls the `shmemx_am_wait` routine and it could not find pending AM requests, it blocks the execution, enters a passive polling mode until an AM request arrives. Both routines could pass another pointer to user-defined arguments as the fourth argument (`args_p`) to the handler, so the handler can have easy access to the calling context.

```
// Active message handler definition
void am_handler(void* payload,
                size_t length,
                void* args_r,
                void* args_p,
                int src_pe)
{
  database_t* db       = args_r;
  scheduler_ctx_t* ctx = args_p;
  int payload_index    = db->store(payload, length, src_pe);
  ctx->insert_task(payload_index);
}

// Initiating PE of the AM request
shmemx_am_send_nbi(am_id, payload, length, pe_id);
shmem_quiet();

// Destination PE
shmemx_am_set_handler(am_handler, &database, &am_id);

while (!scheduler_ctx.done()) {
  shmemx_am_wait(&scheduler_ctx);
  int got_new_am;
  do {
    got_new_am = shmemx_am_poll(&scheduler_ctx);
  } while (got_new_am != 0);
}
```

Listing 1.3. Sample Usage of the Proposed OpenSHMEM Active Message API

The two polling routines can be combined in a fashion that is similar to the adaptive spinlocks: wait is used to put the thread to "sleep" while waiting for incoming AM requests to reduce resource usage; after wake-up, we perform busy non-blocking polling with the other routine until it returns 0, then go back to the blocking wait.

We deliberately choose to not have a background polling thread because we want the developers to have precise control over when and where the AM handlers are executed. This decision is crucial for inter-operating with OpenMP tasks: OpenMP does not provide an "entry" to its task scheduler for inserting new tasks on the fly, the user must write `#pragma omp task` inside the AM handler and make sure the handler is executed by an OpenMP-managed thread to inject a new task into a parallel region.

An example showing how the proposed API extension could be used to insert a task into a hypothetical shared-memory tasking system is presented in Listing 1.3. The active message handler stores the payload and the ID of the initiator PE into a database on the destination PE. Then the handler inserts a task

into the task scheduling context that processed this AM request, so the shared-memory tasking framework can pick it up later and process the corresponding entry in the payload database. The initiator sends the AM request using an `am_id` that is the same as the one returned by the `shmemx_am_set_handler` routine on the destination PE and flushes the default context so the payload buffer can be reused. On the destination PE, we use the wait-and-poll combo to receive AM requests and insert tasks to the current scheduling context, until the work is done.

The example above can also be viewed as a child task sending its execution results to the parent task, or as a parent task sends a unit of work to one of its children for execution. This approach could be extended to execute a dynamic DAG of tasks that changes based on run-time information. Task migration through our active message extension is more flexible and maintainable than allocating one signal variable for each edge in the DAG and perform manual task queue management.

3.2 Implementation

Our implementation is based on the reference implementation of OpenSHMEM, OSSS-UCX [16]. This implementation uses UCX [22] as its communication substrate, which provides unified low overhead access to various vendor-specific communication protocols like InfiniBand Verbs and Cray uGNI. UCX provides a simple active message interface that works as follows: AM handlers are registered on UCP workers, AM requests are sent through UCP endpoints which represent pairs of "linked" workers, and calling the `ucp_worker_progress` routine on the destination process executes incoming active messages. The UCX AM handlers are invoked from the progress context, so trying to perform nested progression by calling `ucp_worker_progress` from within the handler is not allowed, thus prohibiting the handler from tracking the completion of various non-blocking operations. Constraints like this are common in similar frameworks to prevent deadlocks and other issues, with GASNet being another notable example.

Figure 1 shows how a PE processes an incoming AM request and sends another one from the AM handler. Rectangular boxes represent the execution contexts of different UCP workers and AM handlers, and the outermost rounded boxes represent the execution context of the OpenSHMEM runtime library. The `shmemx_am_send_nbi` routine sends the AM request and the accompanying payload to a dedicated UCP worker (`AM_worker`) on the destination PE, along with some metadata, using the UCX AM mechanism and an internal UCX AM handler. When the destination PE calls `shmemx_am_poll`, the OpenSHMEM runtime calls `ucp_worker_progress` on the `AM_worker` to process incoming AM requests. The UCX AM handler simply stores the pointer to the payload and returns `UCS_INPROGRESS` so that the UCX runtime does not deallocate the payload when the handler returns. Then, the OpenSHMEM runtime calls the requested OpenSHMEM AM handler using the ID assigned during registration. Any communication request initiated from the OpenSHMEM AM handler goes through the worker (`DEF_worker`) that handles the default context to avoid accidentally

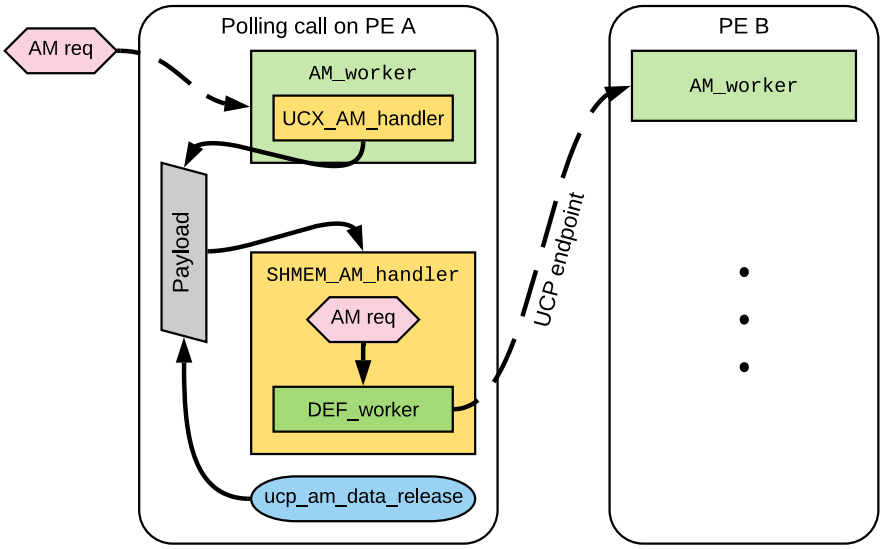

Fig. 1. OpenSHMEM Active Message Implementation in OSSS-UCX.

invoking another incoming AM request. Finally, when the OpenSHMEM AM handler finishes execution, `ucp_am_data_release` is called on the payload buffer to return it to UCX.

This design adds some overhead when compared to UCX active messages, but the two-worker approach enables chaining of AM requests and avoids execution of AM handlers in unexpected places (e.g. quiet, barriers) if the default worker is also used to process incoming active messages.

Advanced features like active message completion notification, automatically return data from the handler to the initiating PE and aggregated active message queues are have been considered for inclusion. These features could be very useful for many applications, but they will increase the complexity of the API and its implementation significantly, so we have decided to not support them in this work. The resulting API is still capable of being used as a task-based programming model and is inter-operable with a shared-memory tasking framework.

4 Performance Evaluation

We perform a preliminary evaluation of the performance of our implementation using two point-to-point micro-benchmarks and the TaskBench [23] framework. The performance numbers presented below were obtained on a cluster equipped with Fujitsu A64FX FX700 CPUs and NVIDIA Mellanox ConnectX-6 100Gb/s network cards. On the software side, the machine is running CentOS 8.1.1911 AArch64, Linux 4.18.0, MOFED 5.0-2.1.8.0, UCX 1.10.1, and GCC 10.3.0. For all MPI results, we use OpenMPI 4.1.1 linked to the same version of UCX.

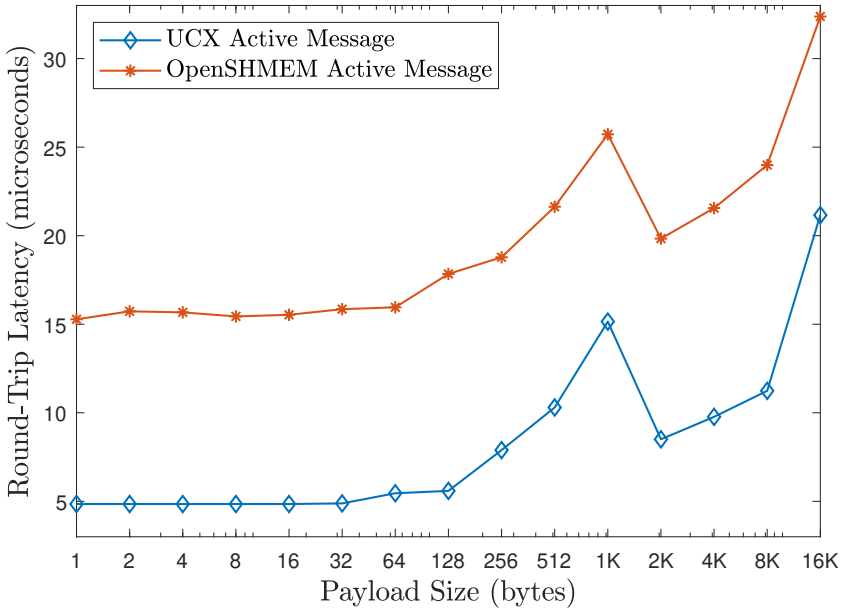

Fig. 2. Active Message Inter-Node Round-Trip Latency Results (UCX protocol switching threshold = 1KiB).

4.1 Latency and Throughput

The classic ping-pong micro-benchmark is a good way to measure the latency of communication operations. A pair of PEs exchange active messages of a certain payload size between two nodes, each side sends one AM request and polls for the other side's AM request to arrive before moving on to the next iteration. We compared the round-trip latency of our OpenSHMEM AM extension against vanilla UCX AM to see how much overhead is added by the implementation shown in Figure 1. The handler only sends an AM request to the other PE so this benchmark does not measure unrelated workload.

From Figure 2, we can see that our implementation adds roughly 10 μs to every round-trip of active messages, so it's 5 μs of overhead per AM request. This overhead is the result of fetching the OpenSHMEM AM handler from the hash table of registered handlers, parsing the AM metadata, and other internal operations. The drop in latency when the payload increases from 1K to 2K is caused by UCX switching its communication protocols. It is worth noting that the two worker approach slightly increases memory usage and slows down the launching of OpenSHMEM applications, but once the application is up and running, the impact should be minimal.

Message throughput benchmark results are shown in Figure 3, where one PE sends a large amount of AM requests with a trivial handler to the other PE and waits for the completion of all the requests. From the numbers, we can see that

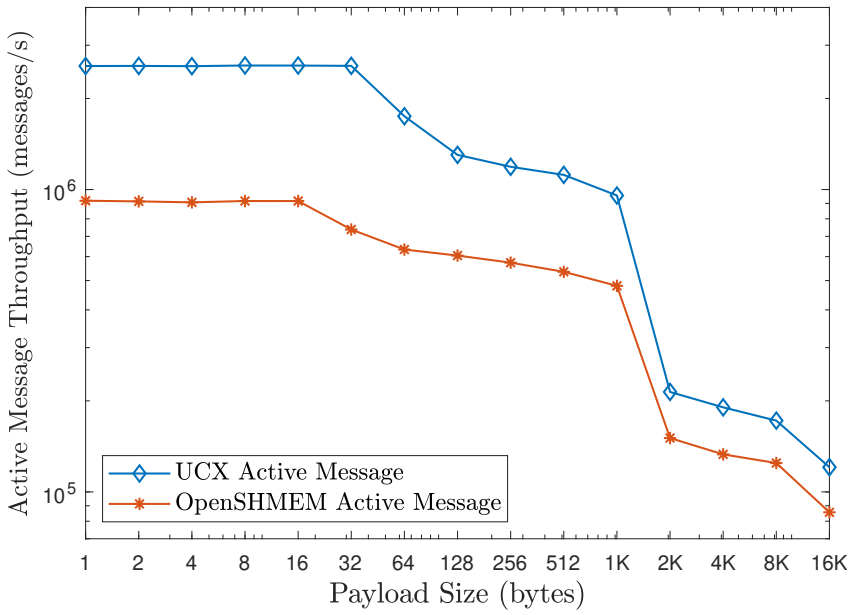

Fig. 3. Active Message Inter-Node Throughput Results (UCX protocol switching threshold = 1KiB).

the message rate of OpenSHMEM AM is consistently lower than that of vanilla UCX AM, but the difference is within a reasonable range and is expected since the OpenSHMEM runtime library performs extra work. Again, the sudden drop of throughput in both implementations is caused by UCX switching protocols.

4.2 Tasking Framework Efficiency

Measuring the performance of a tasking framework is not easy: micro-benchmark results do not translate well to real-world application performance; it is difficult to create comparable ports of mini-apps and above to different tasking frameworks; even strong and weak scalability results can have different measuring methodologies and interpretations. TaskBench [23] is a new benchmark framework designed to provide a better way to compare different ways to do task-based programming. TaskBench makes the process of creating benchmarks (different DAGs and types of workload) orthogonal to the process of adding a new tasking framework backend, thus enabling the comparison of different programming models on equal grounds. Additionally, minimum effective task granularity (METG) is proposed as a new metric to compare the performance of tasking frameworks. METG is defined as the minimum task granularity that can utilize the hardware effectively for a given combination of hardware and workload, with a user definition of what is effective (usually being the ability to reach a certain percentage of the computer's peak performance). It is based on

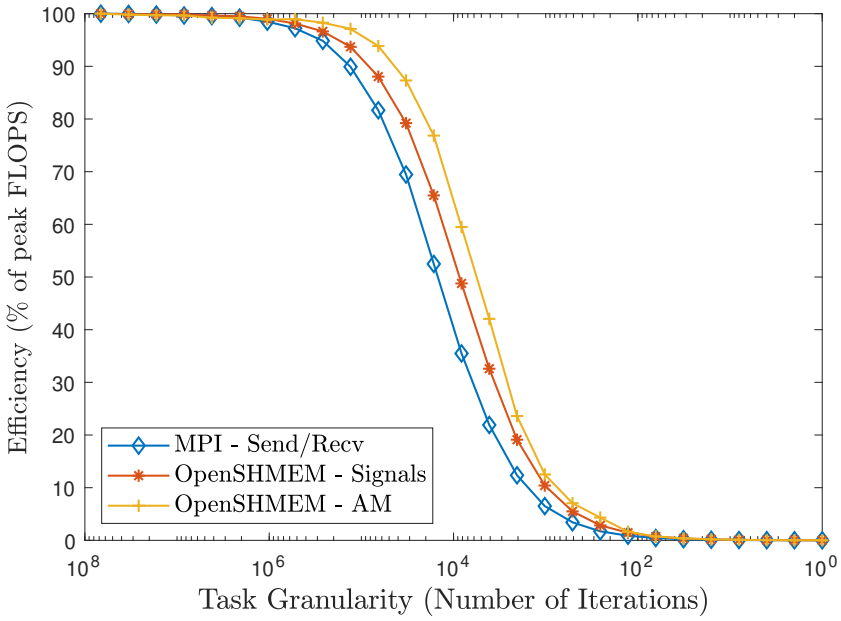

Fig. 4. TaskBench Efficiency v.s. Task Granularity Results.

the assumption that application efficiency drops as task granularity decreases, which is generally true due to the high scheduling cost of the execution of a large task DAG.

To compare the METG of our active message extension against OpenSHMEM's signal variable approach and MPI's message-passing approach, we run TaskBench on 8 nodes with 8 PEs/ranks per node (a total of 64 processes). The DAG used here is a 256×256 array of tasks with the all-to-all dependency pattern and 16 bytes of input/output data along the edges. Each task runs a compute-bound kernel with various numbers of iterations of synthetic computations to simulate tasks of different granularities. We use the MPI backend that comes with the TaskBench package, a signal-and-get OpenSHMEM back-end, and an OpenSHMEM active message back-end that uses AM requests to launch tasks and complete dependencies.

The results are presented in Figure 4. Similar to the original paper's methodology [23], we define the machine's peak performance to be the FLOPS achieved when using a very large task granularity and calculate the efficiency of a particular task granularity by dividing the FLOPS obtained at that granularity by the peak FLOPS. From the achieved percentages of peak FLOPS show in Table 1, we can see that our OpenSHMEM active message backend shows a clear advantage ($6\% \sim 24\%$) between task granularities from 2^{10} to 2^{16} iterations. DAGs with longer tasks are compute-bound and DAGs with shorter tasks are latency-bound, so we see similar results for all three backends. Also, from the

Task Granularity	MPI Send/Recv	OpenSHMEM Signals	OpenSHMEM AM
1024	6.503%	10.41%	12.51%
2048	12.35%	19.11%	23.60%
4096	21.91%	32.59%	42.06%
8192	35.49%	48.78%	59.48%
16384	52.48%	65.49%	**76.85%**
32768	69.46%	**79.22%**	87.32%
65536	**81.66%**	88.05%	93.85%

Table 1. Percentage of Peak FLOPS Achieved at Different Task Granularities

embolden percentages in Table 1, we can see that to reach 80% of the peak FLOPS of our test setup, the active message backend only requires about 1/2 of the task size of the signal-and-get backend, and about 1/4 of the task size of the MPI send/receive backend. This shows that our implementation of the proposed extension provides better performance for a wider range of task-based applications than MPI and classic OpenSHMEM.

5 Conclusion and Future Work

In this paper, we present an OpenSHMEM active message extension and its implementation that is designed to support basic task-based programming on its own, and inter-operates well with shared-memory tasking frameworks. Our proposed API is simple and adheres to the look and feel of existing OpenSHMEM routines. The user is given full control of when and where the AM handlers are executed, and the ability to perform selected communication operations from the AM handlers. The two-worker approach used in our UCX-based prototype implementation separates the initialization and completion of AM requests, which is crucial for the flexibility of the AM handler.

The performance of the implementation of the active message extension was evaluated using point-to-point micro-benchmarks and TaskBench. Evaluation results show that our design adds a reasonable amount of overhead when compared to vanilla UCX AM, and it beats both the MPI back-end and a signal-and-get-based OpenSHMEM back-end of TaskBench in minimum effective task granularity. The proposed active message extension narrows the functionality gap between OpenSHMEM and other task-based programming models.

In the future, we plan to explore the feasibility and performance trade-off of adding optional AM completion notifications, as it could further simplify dependency management of the task DAG. We also plan to port mini-apps like MiniAMR [19] to OpenSHMEM+X where X is a shared-memory tasking model like OpenMP, Intel TBB [18] and Taskflow [9], and evaluate the real-world performance benefits of our active message extension in load-imbalanced applications.

Acknowledgment

This research was funded in part by the United States Department of Defense, and was supported by resources at Los Alamos National Laboratory, operated by Triad National Security, LLC under Contract No. 89233218CNA000001.

The authors would also like to thank Stony Brook Research Computing and Cyberinfrastructure, and the Institute for Advanced Computational Science at Stony Brook University for access to the innovative high-performance Ookami computing system, which was made possible by a $5M National Science Foundation grant (#1927880).

References

1. Bachan, J., Baden, S.B., Hofmeyr, S., Jacquelin, M., Kamil, A., Bonachea, D., Hargrove, P.H., Ahmed, H.: UPC++: A High-Performance Communication Framework for Asynchronous Computation. In: Proceedings of the 33rd IEEE International Parallel & Distributed Processing Symposium. IPDPS, IEEE (2019). https://doi.org/10.25344/S4V88H, https://escholarship.org/uc/item/1gd059hj
2. Bauer, M., Treichler, S., Slaughter, E., Aiken, A.: Legion: Expressing locality and independence with logical regions. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. pp. 1–11 (2012). https://doi.org/10.1109/SC.2012.71
3. Bonachea, D., Hargrove, P.H.: GASNet-EX: A High-Performance, Portable Communication Library for Exascale. In: Proceedings of Languages and Compilers for Parallel Computing (LCPC'18). Lecture Notes in Computer Science, vol. 11882. Springer International Publishing (October 2018). https://doi.org/10.25344/S4QP4W, https://link.springer.com/chapter/10.1007/978-3-030-34627-0_11, lawrence Berkeley National Laboratory Technical Report (LBNL-2001174)
4. Chamberlain, B.L.: Chapel (Cray Inc. HPCS Language), pp. 249–256. Springer US, Boston, MA (2011), https://doi.org/10.1007/978-0-387-09766-4_54
5. Chapman, B.M., Curtis, T., Pophale, S., Poole, S.W., Kuehn, J.A., Koelbel, C., Smith, L.: Introducing openshmem: Shmem for the pgas community. In: PGAS (2010)
6. Daiß, G., Amini, P., Biddiscombe, J., Diehl, P., Frank, J., Huck, K., Kaiser, H., Marcello, D., Pfander, D., Pfüger, D.: From piz daint to the stars: Simulation of stellar mergers using high-level abstractions. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3295500.3356221, https://doi.org/10.1145/3295500.3356221
7. Dinan, J., Flajslik, M.: Contexts: A Mechanism for High Throughput Communication in OpenSHMEM. In: Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. pp. 10:1–10:9. ACM, New York, NY, USA (2014). https://doi.org/10.1145/2676870.2676872, http://doi.acm.org/10.1145/2676870.2676872
8. Eicken, T., Culler, D., Goldstein, S., Schauser, K.: Active messages: A mechanism for integrated communication and computation. In: [1992] Proceedings the 19th

Annual International Symposium on Computer Architecture. pp. 256–266 (1992). https://doi.org/10.1109/ISCA.1992.753322
9. Huang, T.W., Lin, D.L., Lin, Y., Lin, C.X.: Taskflow: A general-purpose parallel and heterogeneous task programming system. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems pp. 1–1 (2021). https://doi.org/10.1109/TCAD.2021.3082507
10. Jana, S., Curtis, T., Khaldi, D., Chapman, B.: Increasing computational asynchrony in openshmem with active messages. In: Gorentla Venkata, M., Imam, N., Pophale, S., Mintz, T.M. (eds.) OpenSHMEM and Related Technologies. Enhancing OpenSHMEM for Hybrid Environments. pp. 35–51. Springer International Publishing, Cham (2016)
11. Kaiser, H., Diehl, P., Lemoine, A.S., Lelbach, B.A., Amini, P., Berge, A., Biddiscombe, J., Brandt, S.R., Gupta, N., Heller, T., Huck, K., Khatami, Z., Kheirkhahan, A., Reverdell, A., Shirzad, S., Simberg, M., Wagle, B., Wei, W., Zhang, T.: Hpx - the c++ standard library for parallelism and concurrency. Journal of Open Source Software **5**(53), 2352 (2020). https://doi.org/10.21105/joss.02352, https://doi.org/10.21105/joss.02352
12. Kale, L.V., Krishnan, S.: Charm++: A portable concurrent object oriented system based on c++. In: Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications. p. 91–108. OOPSLA '93, Association for Computing Machinery, New York, NY, USA (1993). https://doi.org/10.1145/165854.165874, https://doi.org/10.1145/165854.165874
13. Lu, W., Curtis, T., Chapman, B.: Enabling low-overhead communication in multithreaded openshmem applications using contexts. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). pp. 47–57 (2019). https://doi.org/10.1109/PAW-ATM49560.2019.00010
14. NVSHMEM. https://developer.nvidia.com/nvshmem
15. OpenSHMEM Application Programming Interface Version 1.4. http://openshmem.org/site/sites/default/site_files/OpenSHMEM-1.4.pdf
16. Open Source Software Solutions (OSSS) OpenSHMEM Implementation on top of OpenUCX (UCX) and PMIx. https://github.com/openshmem-org/osss-ucx
17. Ozog, D., Rahman, M.W.u., Taylor, G., Dinan, J.: Designing, implementing, and evaluating the upcoming openshmem teams api. In: 2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM). pp. 37–46 (2019). https://doi.org/10.1109/PAW-ATM49560.2019.00009
18. Pheatt, C.: Intel® threading building blocks. J. Comput. Sci. Coll. **23**(4), 298 (Apr 2008)
19. Sasidharan, A., Snir, M.: Miniamr - a miniapp for adaptive mesh refinement (2016)
20. Scherer, W.N., Adhianto, L., Jin, G., Mellor-Crummey, J., Yang, C.: Hiding latency in coarray fortran 2.0. In: Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model. PGAS '10, Association for Computing Machinery, New York, NY, USA (2010). https://doi.org/10.1145/2020373.2020387, https://doi.org/10.1145/2020373.2020387
21. Schuchart, J., Bouteiller, A., Bosilca, G.: Using mpi-3 rma for active messages. In: 2019 IEEE/ACM Workshop on Exascale MPI (ExaMPI). pp. 47–56 (2019). https://doi.org/10.1109/ExaMPI49596.2019.00011
22. Shamis, P., Venkata, M.G., Lopez, M.G., Baker, M.B., Hernandez, O., Itigin, Y., Dubman, M., Shainer, G., Graham, R.L., Liss, L., et al.: Ucx: an open source framework for hpc network apis and beyond. In: 2015 IEEE 23rd Annual Symposium on High-Performance Interconnects. pp. 40–43. IEEE (2015)

23. Slaughter, E., Wu, W., Fu, Y., Brandenburg, L., Garcia, N., Kautz, W., Marx, E., Morris, K.S., Cao, Q., Bosilca, G., Mirchandaney, S., Lee, W., Treichler, S., McCormick, P., Aiken, A.: Task bench: A parameterized benchmark for evaluating parallel runtime performance. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. SC '20, IEEE Press (2020)
24. Zhao, X., Balaji, P., Gropp, W., Thakur, R.: Mpi-interoperable generalized active messages. In: 2013 International Conference on Parallel and Distributed Systems. pp. 200–207 (2013). https://doi.org/10.1109/ICPADS.2013.38