ActorProf: A Framework for Profiling and Visualizing Fine-grained Asynchronous Bulk Synchronous Parallel Execution

Jiawei Yang[§1], Shubhendra Pal Singhal[§2], Jun Shirako[3], Akihiro Hayashi[4], Vivek Sarkar[5]
{[1]jyang810, [2]ssinghal74, [3]shirako [4]ahayashi, [5]vsarkar}@gatech.edu
Georgia Institute of Technology
Atlanta, GA, USA

***Abstract*—A Fine-grained Asynchronous Bulk Synchronous Parallel (FA-BSP) model is an extended version of the existing BSP model that facilitates fine-grained asynchronous point-to-point messages with automatic message aggregation.**
While many large irregular applications written with the FA-BSP model demonstrate promising performance, no profiler is aware of profile-worthy portions of an FA-BSP program and visualizes the results in an intuitive way. This is reasonable because the FA-BSP program relies on multiple external libraries, and the runtime frequently switches between different portions of the program, which makes it difficult for well-established profilers like `score-p`, `TAU(`τ`)`, `CrayPat`, `Intel®Vtune™`, and `HPCToolkit` to profile and visualize these portions in an FA-BSP-friendly manner.
This paper designs and implements a profiling and visualization framework called `ActorProf`. The framework enables 1) asynchronous point-to-point message-aware profiling with hardware performance counters, 2) overall performance breakdown that is aware of FA-BSP execution, and 3) visualization of these profiling results.

***Index Terms*—ActorProf, HClib, FA-BSP, SPMD, Actors, Selectors, Conveyors, OpenSHMEM**

I. Introduction

In the exascale era of computing, irregular applications like Breadth First Search (BFS), Triangle Counting, and PageRank built on top of the Bulk Synchronous Parallel (BSP) model [1] such as OpenSHMEM [2] and MPI [3] face a common challenge: *"sending large orders of small byte-sized messages (is generally of ∼8-32 bytes for billion in number) degrades performance due to the under-utilization of the network bandwidth"* [4]. This problem can significantly limit large-scale irregular applications' ability to scale effectively in both strong and weak scaling scenarios. One solution to the problem is using *Message Aggregation*, where multiple small messages intended for the same destination are combined into a larger message to improve bandwidth utilization.

Conveyors is one of the state-of-the-art message aggregation libraries [4]. While it shows promising scalability for different large-scale irregular applications, it poses another orthogonal issue: the programming is complex. In Conveyors programs, users need to manually interleave the message sending and receiving parts in the same portion of the code and also handle errors such as aggregation buffer overflow in that portion. This process is error-prone and typically hinders productivity. The Fine-grained Asynchronous Bulk Synchronous Parallel (FA-BSP) model [5] is introduced to abstract away such complexity of message aggregation, while enabling promising scalability. In a nutshell, the FA-BSP model can be viewed as an extended version of the BSP model that facilitates fine-grained *asynchronous* point-to-point messages with automatic message aggregation without any user-written error handling. As illustrated in Figure 1, each processing element (PE), performs 1) a local computation (the BLUE part), 2) asynchronous messaging (the arrows) during the local computation, and 3) message handlers (the RED part) in an interleaved fashion.

[§]These authors contributed equally to this work.

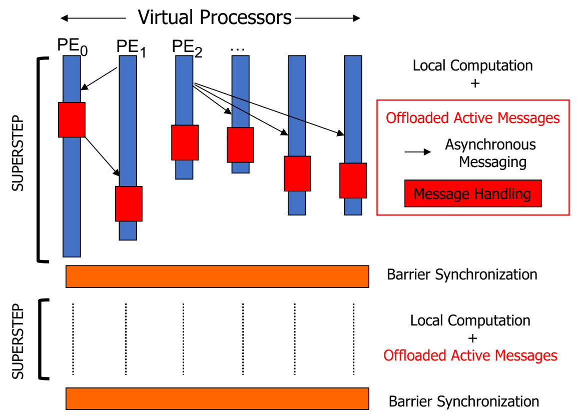

Figure 1: Fine-grain Asynchronous Bulk Synchronous Parallel Model (FA-BSP) [5]

HClib-Actor [6] is the only realization of the FA-BSP model as of this writing. In the literature, many irregular applications are written with HClib-Actor, demonstrating promising strong/weak scaling and even outperforming the state-of-the-art counterparts ([7]–[9]).

However, one challenging problem with HClib-Actor programs is profiling and visualizing meaningful data for the user. From the users' perspective, they want a profiler to recognize

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00199

and profile the local computation, asynchronous messaging, and message handlers parts. Because the HClib-Actor relies on lower-level libraries such as Conveyors and OpenSHMEM for asynchronous communication and message aggregation, as well as HClib for interleaved execution, this presents a challenge for existing profilers such as `score-p`, `TAU(`τ`)`, `CrayPat`, and `Intel®Vtune™` to profile and visualize these parts in an FA-BSP-friendly manner.

To alleviate this problem, this paper designs and implements a profiling and visualization framework called `ActorProf`. This paper makes the following contributions:

1) Message-aware Profiling for HClib-Actor: `ActorProf` generates statistics on 1) point-to-point sends between PEs before aggregation (what we call *Logical Trace*), and 2) `PAPI` [10] counter numbers for user-given PAPI events for each local computation and message handling part. (Section III-A)
2) Overall Breakdown for HClib-Actor: `ActorProf` also measures how many `rtdsc()` cycles are spent in the three key parts of an HClib-Actor programs: the local computation part, the message handling part, and the communication part. This gives the user a big picture of their HClib-Actor program execution. (Section III-B)
3) After Aggregation Message Profiling for Conveyors: `ActorProf` also generates statistics on point-to-point sends between PEs after aggregation (what we call *Physical Trace*), which is supposed to be done by existing profilers because aggregated messages are sent via OpenSHMEM's non-blocking routines. However, it is unfortunate that existing profilers can not correctly capture such non-blocking routines [11], and we decided to generate physical traces. (Section III-C)
4) Data Visualization for HClib-Actor, which helps the user infer performance bottlenecks. (Section III-D).

`ActorProf` can be downloaded from https://github.com/srirajpaul/hclib/tree/bale3_actor/modules/bale_actor/tools.

We organize this paper into six sections. Section II explains the FA-BSP model for communication and the integration with Conveyors and OpenSHMEM. Section III introduces the system design of `ActorProf`, which records multiple information depending on the macros defined and provides the internal working of the tool as a guide to profile asynchronous communication tasks on multi-node distributed architectures. In Section IV, we illustrate the tool's usage by analyzing and showing experiments for the Triangle Counting application. We conclude the paper by discussing the related and future work in Section V and Section VI respectively.

II. Background

This section briefly discusses the FA-BSP model. For the terminology used in the paper, refer to Table I.

The Habanero C/C++ library (HClib) [12] was developed to enable an asynchronous many-task (AMT) programming model and its runtime system on a single-node platform. It inherits different parallel constructs originally from the X10 [13] language.

TABLE I: Terminology used in this paper

Abbreviations	Description
`PGAS`	Partitioned Global Address Space
`SPMD`	Single Program Multiple Data
`BSP`	Bulk Synchronous Parallel
`FA-BSP`	Fine-grained Asynchronous Bulk Synchronous Parallel
`MAIN`	Segment of constructing a message and any local computation (The body of `finish` minus `send` in Listing 1)
`PROC`	Segment of message handling (`process` in Listing 2)
`COMM`	Any region outside `MAIN` and `PROC`
`HWPC`	Hardware Performance Counters
`PE`	OpenSHMEM processing element. There is one actor instance per PE.
`Node`	A cluster node, group of PEs
`Selector`	HClib-Actor with multiple mailboxes
`Segment`	Culmination of functions which does not involve any asynchronous communication
`Region`	Placement of trace APIs are only applicable to use in HClib-Actor. The user application is prohibited from accessing these APIs.

HClib-Actor [6] is an external PGAS-inspired (Partitioned Global Address Mailbox) actor-based [14] module for HClib that enables the FA-BSP execution on a multi-node platform. The features of HClib-Actor include (1) Asynchronous messaging with actor/selector (Selector [15] is an actor with multiple mailboxes), (2) SPMD-style programming with OpenSHMEM [2], and (3) Automatic message aggregation with the Conveyors library [4].

A. HClib-Actor

Listing 1: The MAIN Part (BLUE part and → in Figure 1)

```
// SPMD
int* larray = (int*)calloc(N, sizeof(int));
MyActor* actor_ptr = new MyActor(larray);
hclib::finish([=]() {
    actor_ptr->start();
    for (int i = 0; i < N; i++) {
        int dst = ...;
        // Asynchronous SEND
        actor_ptr->send(i, dst);
    }
    actor_ptr->done(0);
});
```

Listing 2: The PROC Part (RED part in Figure 1)

```
// Actor Class
class MyActor: public hclib::Selector<1, int> {
    int *larray;
    // Message Handler
    void process(int idx, int sender_rank) {
        larray[idx] += 1; // no atomics
    }
public:
    MyActor(int *larray) : larray(larray) {
        mb[0].process = [this](int idx, int sender_rank) {
            this->process(idx, sender_rank);
        };
    }
};
```

Listing 1 and Listing 2 demonstrate an FA-BSP program with HClib-Actor. In this program, each processing element in OpenSHMEM (coined as PE) sends N messages to arbitrary destinations, incrementing a target element of a remote array

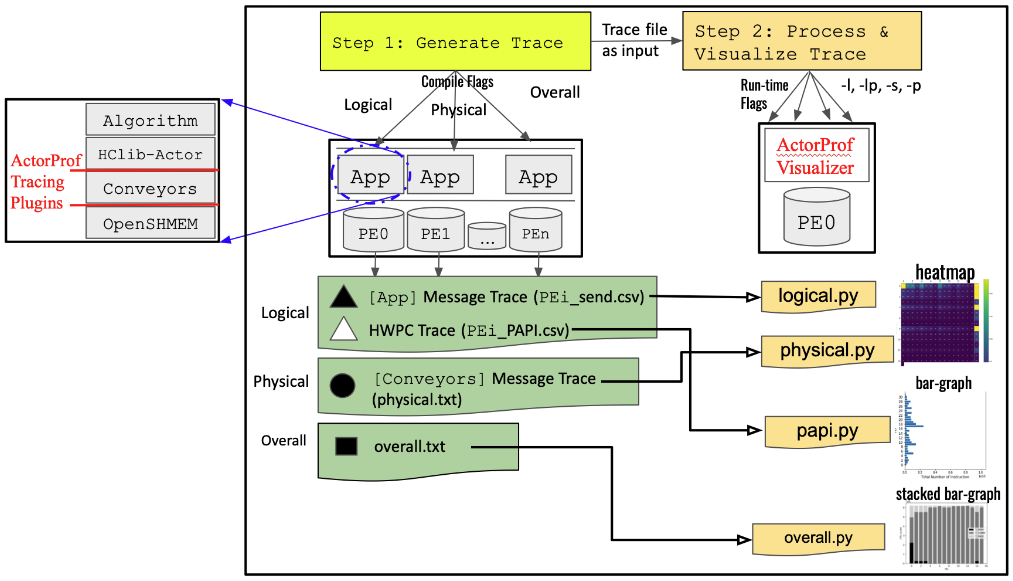

Figure 2: System Design of ActorProf

by one. In Listing 1, each PE first allocates a local array larray (Line 2). Second, each PE instantiates an actor instance (Line 3). Third, each PE starts the actor (Line 5) and sends N asynchronous messages to random destinations (Line 9). The done API (Line 11) informs the runtime that the current PE will not send any more messages to aid the runtime with overall application termination. The code in Listing 2 defines an actor class that includes the message handler (Line 5). It is important to note that no atomics are required on Line 6 when updating larray because the runtime processes incoming messages one at a time. Each PE is single-threaded. The finish construct is used to wait until all outgoing messages are sent and all incoming messages are processed.

B. *Conveyors*

Conveyors is a standalone library that can be built on top of either OpenSHMEM, MPI, or UPC. It aims to address the single-threaded producer-consumer problem by developing a push-style aggregation policy with multi-hop routing, making it memory-frugal and exploiting better network bandwidth utilization. The adoption of one-sided puts in a performant manner was shown in 2019 by Conveyors on OpenSHMEM by overcoming the bottlenecks of past libraries that attempted to perform aggregation - exstack (global synchronization problem), exstack2 (memory scalability of buffers), TRAM from Charm++ [16] (does not support intra-node network optimizations), and YGM [17](restricted to only one mailbox). Adoption of Conveyors by run-time system HClib as HClib-Actor in 2022 offered FA-BSP programming model, thereby enabling nesting of Conveyors objects such that multiple Conveyors objects can communicate with each other using a partitioned mailbox. Note that Conveyors is just an aggregation library that does not have a binding to any programming model.

III. System Design and Implementation

The end-to-end system design of ActorProf is illustrated in Figure 2.

For ActorProf "trace-collection": The user application in a multi-node environment requires compile flags to be enabled, as listed below:

- -DENABLE_TRACE for logical trace (Section III-A)
- -DENABLE_TCOMM_PROFILING for Overall profile (Section III-B)
- -DENABLE_TRACE_PHYSICAL for physical trace (Section III-C)

ActorProf begins the trace generation by using tracing hooks placed inside run-time system HClib-Actor, and the aggregation library Conveyors. Software trace APIs record source, destination, buffer/message sizes (in bytes), and local timestamp if needed. HWPC API's record various architecture dependent PAPI counters and utilize the region-specific features: PAPI_start and PAPI_stop to record during asynchronous communication.

ActorProf "visualization": This step requires the completion of "trace generation." ActorProf generates heatmap and different styles of bar-graph. The user is required to run ActorProf with run-time flags:

- -l for logical trace (▲) heatmap.
- -lp for PAPI trace (△) bar graph for four PAPI counters in one run.
- -s for Overall trace (■) stacked bar graph for both absolute and relative execution time.
- -p for physical trace (●) heatmap.

A. Logical Trace

`Logical trace` (▲) records the "user application-fed" source and destination records. Moreover, it also records the logical mapping between nodes/PEs/threads of the cluster and actor. **HPC algorithm designers** can comprehend the communication pattern based on the respective input/internal data-structure distributions.

Refer to the terminologies in Table I. `HWPC` in `ActorProf` (△) provides overall profiling of segments of a user application, where a user can utilize HClib-Actor tracing functions. Segments refer to the culmination of functions that do not involve any asynchronous communication. In addition, `ActorProf` also performs region-specific profiling for two regions - `MAIN` is the segment of constructing a message & any local computation, and `PROC` is the segment responsible for the message handling/`recv` part. `ActorProf` only allows up to four concurrent recording events with the limitation from `PAPI`. This form of profiling helps the user separate the measurement of the counters during the context switch between the `send` and the `recv` task.

This feature benefits both **HPC algorithm designers** and **HPC run-time designers**, since the former can experiment and deduce the resource bottleneck bounds of their implementation and suitability of the run-time system HClib-Actor from the perspective of FA-BSP model, and the latter, can obtain the suite/common patterns of algorithms that could potentially degrade their system's performance, as on degradation for instance, memory (data and instruction) counters in `PAPI` indicate cache/TLB thrashing; information on loads/stores and branch prediction stalls; data prefetch cache misses counters - a concept highly useful for exploiting overlap; retired instruction profiling; Vector/SIMD profiling if user-application such as in [18] utilizes the vectorization feature.

Implementation: This feature creates two types of files per PE, `PE`i`_send.csv` and `PE`i`_PAPI.csv`, where each line of the file represents the trace information of a single send operation in the following formats:

- `PE`i`_send.csv`: Logical trace for PE i.
  - `source node, source PE, destination node, destination PE, message size`
- `PE`i`_PAPI.csv`: PAPI-based message trace for PE i.
  - `source node, source PE, dst node, dst PE, pkt size, MAILBOXID, NUM_SENDS, PAPI_TOT_INS, PAPI_LST_INS`

We used `PAPI_TOT_INS` as one of the `PAPI` counters in the case study, which records the total number of instructions within the region specified.

B. Overall profiling

This profile (■) aims at distinguishing the time taken solely due to communication from the time taken by "user-provided" `send/recv` functions. Refer to the terminologies: `MAIN` and `PROC` in Table I. HClib primarily involves three sub-parts: (1) T_MAIN, The time taken by the application to generate a message and append it to the mailbox; (2) T_COMM, The time taken in communication (derived); and (3) T_PROC, The time taken by user-provided message handler to process the pulled messages. We derive (2) by `Total-(1)-(3)`, which is solely responsible for communication by Conveyors. We denote the total time taken by T_TOTAL. This feature of profiling guides the **HPC programmers** to optimize their implementations in case (1) and (3) are observed as a bottleneck, else **HPC algorithm designers** need to brainstorm the opportunity to exploit more overlap between computation and communication.

Implementation: This feature generates a `overall.txt` file where each line of the file indicates the absolute and relative time of a single `Selector` in the following formats.

- `overall.txt`: physical message trace for all PEs.

Absolute `[PE`i`] TCOMM_PROFILING (T_MAIN, T_COMM, T_PROC)`

Relative `[PE`i`] TCOMM_PROFILING (T_MAIN/T_TOTAL, T_COMM/T_TOTAL, T_PROC/T_TOTAL)`

Note that `rdtsc` x86-instruction is used to record the time.

C. Physical Trace

`Physical trace` (●) records the network-fed route source and destination traces which are dictated by the topology of Conveyors- 1D Linear/2D Mesh/3D Cube topology [11] [4], which consists of a *static/fixed* route of communication for a pair of source and destination. `ActorProf` traces the following communication calls in Conveyors:

- `local_send`: All PEs on the same node (Intra-Node) perform std::memcpy using shmem_ptr.
- `nonblock_send`: Inter-Node send is initiated and marked for send by the network layer uses non-blocking shmem_putmem_nbi.
- `nonblock_progress`: Inter-Node completion of sends to all destinations is mandated for the sender PE such that the data sent is visible to all PEs. It uses shmem_quiet followed by shmem_put for signaling the destination PE.

This feature enables fine-grain profiling for the FA-BSP model (i.e., HClib-Actor + Conveyors), which benefits both **HPC algorithm designers** and **HPC run-time designers**, since the former can experiment and deduce the suitability of the FA-BSP model, and the latter can obtain number/patterns of the network communication that could potentially degrade their system's performance.

Implementation: This feature generates a `physical.txt` file where each line of the file indicates the single send operation recorded from instrumentation in Conveyors, in the following formats:

- `physical.txt`: physical trace for all PEs.
  - `send type, buffer (network-packet) size, source PE, destination PE`

Note that the `send type` includes `local_send`, `nonblock_send`, and `nonblock_progress`.

D. Trace Visualization

ActorProf Visualization utility is inspired by CrayPat's feature - "Mosaic Report," which depicts the matrix of communication between the source and destination PEs, using the colored blocks based on the number of sends. Our utility extends further and offers insights into the relative performance of time, and HWPC for every PE, load imbalance of "actors" due to application sends (wrt to actor PEs), and hotspots of "node" from the network sends.

We offer various visual interpretations, such as Heatmap, Quartile Violin plot, and Bar Graph with additional support for stacked columns. Bar Graphs can be used to identify the stragglers in the system and study trends of time/counters for all or user-specific regimes of HClib-Actor programs. Furthermore, heatmaps showcase communication patterns (for both virtual and network s/w library topology) along with the number of sends between each pair of source and destination and total outgoing send/recv for every PE, represented in the last row and the last column. Violin plots are used for inferring the quartiles for total send/recv traces.

Implementation: This feature generates different graphs for different types of traces using python scripts and inputs as follows:

- logical.py: Generate heatmap from logical message trace file PEi_send.csv
- physical.py: Generate heatmap from physical message trace file physical.txt
- papi.py: Generate bar graph from papi message trace file PEi_PAPI.csv
- Overall.py: Generate bar graph from message trace file Overall.txt

ActorProf Visualizer use python modules: numpy, pandas and matplotlib. The input data format is .txt for physical.py / Overall.py and .csv for logical.py / papi.py. Note that all graph generation scripts take the path to the corresponding trace file(s) directory as a positional argument and the total number of PEs used num_PEs as additional input argument.

IV. Case Study: Distributed Triangle Counting

A. Usage

We are actively using ActorProf in our workloads, to name a few - Influence Maximization [19], Jaccard Similarity [7] etc., for profiling and performance bottleneck identifications. More demonstrating examples can be found at https://hclib-actor.com.

B. Purposes

The purpose of this study is to 1) discuss a scenario where a choice of data distribution of vertices causes load imbalance due to the power law distribution nature of an input R-MAT graph, and to 2) discuss how ActorProf helps the user identify such load imbalance.

1) Distributed Triangle Algorithm: Triangle Counting is a classic graph application that counts all possible numbers of triangles in a graph. Algorithm 1 shows an FA-BSP implementation. The input for the algorithm is the lower triangular part of a global adjacency matrix (L) for a graph. The output is the total number of triangles in the graph. Each actor ($Actor_p$ on rank r_p) iterates over the neighbors of each local vertex i and identifies two distinct neighbors (edges l_{ij} and l_{ik} s.t. $k < j$). To check if there is an edge l_{jk}, it sends a non-blocking message to the (possibly) remote actor that owns an edge l_{jk}. Upon receiving the message and if such an edge exists, a local triangle count on that rank will be incremented by 1.

2) Data Distributions: A data distribution decides which data resides on which rank. Two distributions were used in this experiment: 1D Cyclic and 1D Range. 1D Cyclic uniformly distributes the rows so that each rank has a similar number of vertices, whereas 1D Range non-uniformly distributes the rows to make each rank have a similar number of edges.

In this algorithm, each actor performs $\mathcal{O}(N^2)$ sends for each local vertex, where N is the number of its neighbors. Depending on an input graph and data distribution, a heavy load imbalance can occur in terms of the number of sends per PE.

C. Experimental Setup

We conducted experiments on the NERSC Perlmutter supercomputer at Lawrence Berkeley National Lab [20]. Each CPU node of Perlmutter consists of dual-socket AMD EPYC 7763 (Milan) processors with 64 physical cores per socket, 512 GB of memory, and a single network card connected to an HPE Cray Slingshot 11 network. Cray-SHMEM modules, i.e., cray-openshmemx/11.5.8 and cray-pmi/6.1.10, CC compiler, were used during the experiment and trace data was placed on the Lustre filesystem for efficient I/O reads. The input graph is a lower triangular undirected, unweighted matrix generated on a scale of 16 with R-MAT parameters of $A = 57.0$, $B = C = 19.0$, $D = 5.0$, and an edge factor of 16, following graph500 benchmark standards [21]. We compared the load-balance performance between the 1D Cyclic and 1D Range versions while keeping all other variants below the same. Both versions take the same input graph on 1/2 node with 16/32 PEs. We have validated the experiments by using assertion, which verified [22] the number of triangles obtained by the application with the theoretical answer, also calculated by the application.

D. Preliminary Results

We only profile the primary part of the application, i.e., the Triangle Counting kernel, and leave the reading of the graph and validation of results out of the scope of this section. The heatmap in ActorProf can yield three points of observation:

- Observe the number of sends in the heatmap and observe the last column/row, indicating the total send/recv per PE across all destination PEs.

Algorithm 1 Triangle counting using actors (SPMD).

```
1: function COUNTTRIANGLESACTOR($L$)
2:     Let $L$ be the lower triangular part of an adjacency matrix. $l_{ij} \in L$ indicates whether there is an edge from vertex $i$ to vertex $j$ s.t. $j < i$. $l_{ij} = 1$ means there is an edge.
3:     Let $r_p$ denote the local process's rank as a 1-D coordinate in a logical $0 \dots p-1$ array.
4:     Let $L_p$ be the local rows of $L$ owned by $r_p$, $c_p$ be the local counter owned by $r_p$, and $Actor_p$ denote the actor instance that is running on $r_p$.
5:     for $\{l_{ij}, l_{ik} \in L_p \mid l_{ij} = l_{ik} = 1, k < j < i\}$ do    ▷ Find two distinct neighbors of vertex $i$
6:         Let $pe \leftarrow$ FINDOWNER($l_{jk}$)    ▷ Find a rank that owns $l_{jk}$
7:         $Actor_p.send(pe, j, k)$    ▷ Send an active message (non-blocking)
8:     WAIT()    ▷ Wait until outgoing messages are sent and incoming messages are processed
9:     return ALLREDUCE($c_p$)
10: function ACTORPROCESS($j, k$) on $r_p$    ▷ The message handler: $j$ is row number, and $k$ is col number
11:     if $\{l_{jk} \in L_p \mid l_{jk} = 1\}$ then $c_p$ += 1
12: function FINDOWNER($l_{jk}$)    ▷ Returns a rank that is responsible for row
13:     return j % $p$    ▷ 1-D Cyclic distribution in this case
```

- Observe the violin plot that indicates the quartile of send/recv trace across PEs. The application has good load balance as the plot tends to be fatter and vice versa.
- Observe the distribution of communication, i.e., the shape of the overall matrix.

Logical Trace Heatmap In this section, we first discuss the communication load and its distribution. Then, we explain and reason the shape/pattern of communication.

Figure 3 and Figure 4 shows the number of "messages" sent by a source PE (row-id) to the destination PE (column-id) for 1D Cyclic and 1D Range data distribution for one node and two nodes respectively. For 1D Cyclic distribution for both the heatmaps, PE0 incurs more communication with a specific set of PEs ($\sim$ 3-4 in number) relative to the rest of the pair of PEs. Further, we show the Quartile using violin plots for the total number of sends/recvs in Figure 5. The violin plot shows 1) the distribution of the data with the density plot (colored shape) where wider sections indicate higher data density, 2) the median in a white dot, and 3) the maximum outlier with the farthest point on the top of the colored shape. In one node, the maximum 1D Cyclic recv is 33% more than the sends, and in two nodes, the maximum send is $\sim$ 200% more than the maximum recv. This clearly indicates an imbalance in the total sends/recvs across PEs.

Contrasting and comparing 1D Cyclic with 1D Range, we observe that only a few PEs $\sim$3-4 incur a large number of recvs, while sends are relatively fairly distributed. This observation can be corroborated by Figure 5 range worded x-axis, whose outlier of sends are relatively less than or almost equal to recvs. Further, in comparison, 1D Cyclic performs a maximum of $\sim$6x sends and $\sim$2x recvs.

Therefore, although 1D Range alleviates the problem of load imbalance in sends, it does not eliminate the problem of load imbalance. We encourage users from here now to try more distributions, such as Edge Cut, Cartesian Vertex-Cut, and many more [23]. And definitively, 1D Range is a better-suited distribution than 1D Cyclic for "*fairly-distributed*" communication load.

In addition, the 1D Cyclic version has its communication distributed irregularly across all PEs, as expected, whereas the 1D Range has a lower triangular(**L**) shape. We coin this observation in 1D Range as (**L**) observation.

The goal of range distribution is to "evenly" distribute edges amongst PEs. We take an example to demonstrate the scenario: Suppose PE0 is responsible for the rows of input matrix indexed from [0... i]. Since the input matrix is lower triangular, rows [0... i] will have the non-zero entries (#nnz) in the columns indexed from [0... i] such that row[0] can have atmost 1 #nnz, row[1] can have atmost 2 #nnz $\cdots$ row[i] can have atmost (i+1) #nnz. Similarly, suppose PE1 is responsible for rows of input matrix indexed from [(i+1)... j], edges with the column indices of [0... i] belong to PE0, edges with the column indices [(i+1) ... (j-1)] belong to PE1 and so on. Note that $i, j \dots$ are chosen such that PEs have an equal number of #nnz.

Therefore, if PE1 has to communicate, it only communicates with PE0 or PE1. A similar extrapolation for any q, PEq would store edges in portions which belong to PE0$\dots(q-1)$, as shown in the Figure 6. Since the triangle counting application traverses the neighbor list and sends a message to its neighbor, we validate the observation of the lower triangular(**L**) shape's communication in the heatmap. In addition, Figure 6 also suggests a "very likely" situation such that the total number of incoming communications by PE0 $>$ PE1 $>$ PE2 and so on. Observe the "*monotonically decreasing fashion*" recvs, i.e., the last row of heatmaps in Figure 3 and Figure 4.

Conclusion: The heatmap for 1D Cyclic data distribution clearly shows the heavy load-imbalance in terms of both sends/recvs. While 1D Range data distribution mitigates the load imbalance by sends, the load imbalance by recvs still persists. Therefore, the Logical Trace Heatmap helps users examine and devise better-suited distributions.

Physical Trace Heatmap

Figure 8 and Figure 9 show the Conveyors dictated number

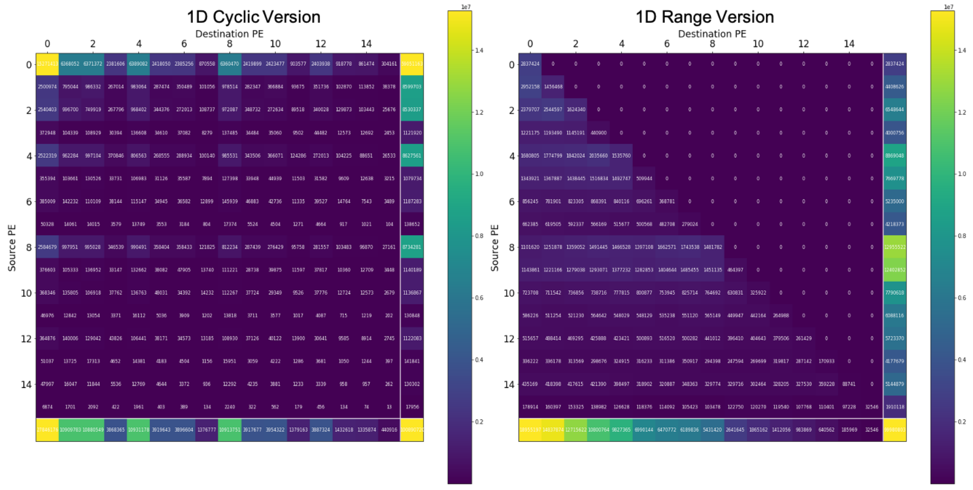

Figure 3: Logical Trace Heatmap for 1 node
(LHS: 1D Cyclic, RHS: 1D Range)

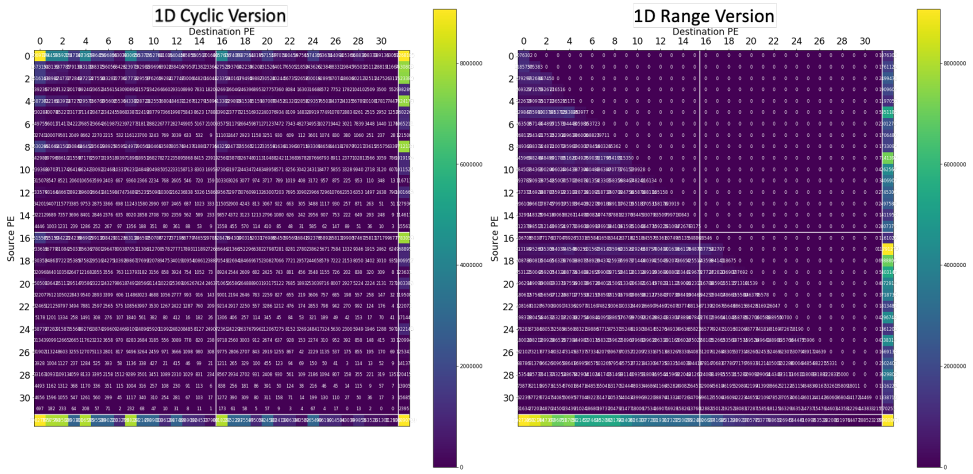

Figure 4: Logical Trace Heatmap for 2 nodes
(LHS: 1D Cyclic, RHS: 1D Range)

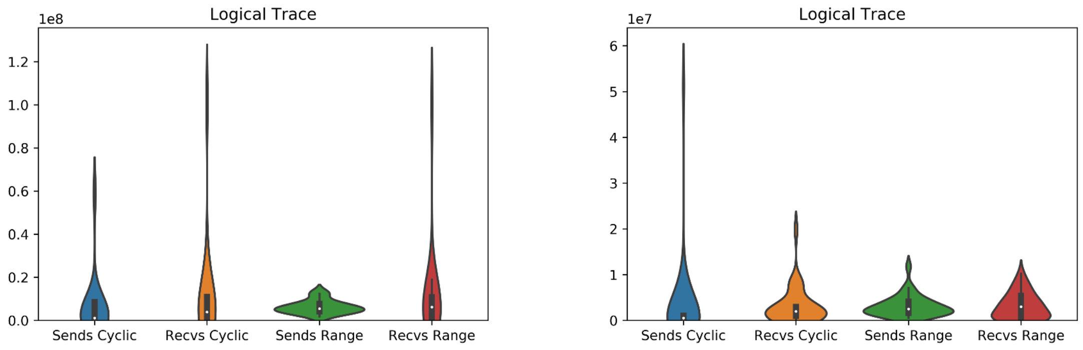

Figure 5: Violin plot for Logical Trace
(LHS: 1 node, RHS: 2 nodes)

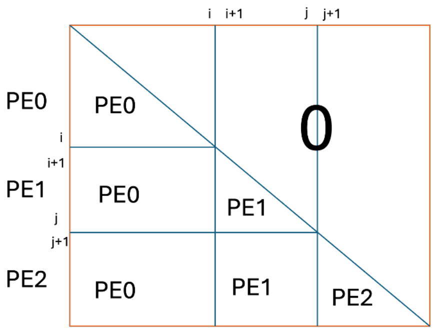

Figure 6: **(L)** observation. Note that $i, j \ldots$ are chosen such that PEs have an equal number of #nnz and should not be confused with the drawing scale shown. The '0' in bold represents #nnz = 0

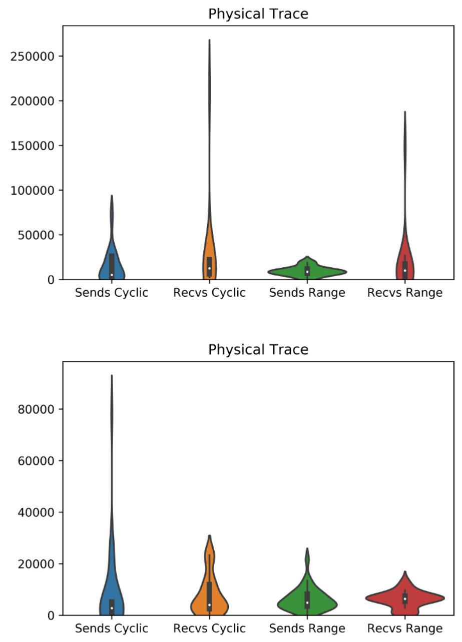

Figure 7: Violin plot for Physical Trace (UP: 1 node, DOWN: 2 nodes)

of "buffers" sent by a PE to the other/destination PE for 1D Cyclic and 1D Range data distribution. Figure 7 indicates the quartile violin plot for physical trace for 1D Cyclic and 1D Range on 1&2 nodes. Sends in 1D Cyclic are worse than those of 1D Range by ~2-4x. Similarly, recvs in 1D Cyclic are worse than those of 1D Range by ~5-15%. 1D Range can still hold a spike, marginally smaller than 1D Cyclic recv. Therefore, 1D Range is an incomplete solution to the overall load-imbalance problem. This is indeed what we observed and concluded for the Logical Trace aforementioned.

Conveyors for one node follow 1D Linear topology, and for two nodes follow 2D Mesh topology, where every PE is restricted to communicate with its row and column member PEs. In mesh topology, PEs use `local_send` along the row and `nonblock_send` along the column. The shape of the heatmaps (1&2 nodes) for `local_send` and `nonblock_send` in 1D Cyclic reflects the underlying topology. For 1D Range, it reflects the **(L)** observation.

Note for self-sends: Excluding this subsection, self-sends have been considered throughout the paper.
One may expect that Conveyors should treat a self-send special by bypassing the network stack. In other words, a maximum of two data copy operations should be incurred, one for copying from the user application to Conveyors `send` and the other for copying back from Conveyors `recv` to the user application. However, such nuanced treatments are not as trivial to implement as they seem. Some algorithms may require a sense of order for their arrival messages to guarantee approximation bounds. For such a space of algorithms, any bypass for self-sends will likely cause "out-of-order" execution. Therefore, `ActorProf` avoids confusion (such as conclusions drawn in Figure 3 with (0,0)) and collects the profile information of sends with/without self separately - `Logical` trace with self-sends and `Physical` without self-sends.

Conclusion: Stays the same as the logical trace. Moreover, an interesting angle originates for **HPC run-time designers** to improve their design and gain more performance; Conveyors can incur up to six `std::memcpy` ops for a single self-send before being delivered to itself, stated in [11] for the abovementioned reason.

`PAPI` Bar Graph

Figure 11 and Figure 10 compare the total number of instructions sent per PE for 1D Cyclic[1] and 1D Range data distribution. In this experiment, we instrument the regime of user-provided code and exclude the Conveyors and HClib-Actor system from the measurement by carefully placing `start` and `stop` `PAPI` APIs. 1D Cyclic version for both one and two node experiments, show that `PE0` suffers from an imbalance (upto ~5x) in the number of instructions compared with other PEs. This is attributed to the last row/column from Figure 3 and Figure 4, where PE0 shows the maximum `sends/recvs` amongst all PEs, indicating the *the measurement for user-code executed in send/recv* to be the maximum on PE0.
Conclusion: PE0 for 1D Cyclic has an imbalance in the total number of instructions, with the worst-case observed up to ~4-5x. Similarly, the measurement of other available `PAPI` counters can help gain such inferences, whose documentation is a continual work-in-progress (inspired from `Intel`®`Vtune`™ official documentation on HPC) and is scoped as the future work.

Overall stacked bar graph

Figure 12 and Figure 13 compare the time taken by regimes, i.e., `MAIN`, `PROC` and `COMM` for HClib-Actor application. `COMM` regime is the bottleneck for both 1D Cyclic and 1D Range. For both setups of different nodes, 1D Range distribution performs ~2x better in total time, as it takes ~300k cycles, while 1D Cyclic takes ~600k cycles. Further, `MAIN` constitutes

[1] In 1D Cyclic, a few PEs do not show any value on the x-axis due to relatively small quantities. The minimum is roughly three to four orders of magnitude lower than the highest, but they are not absolute zeros.

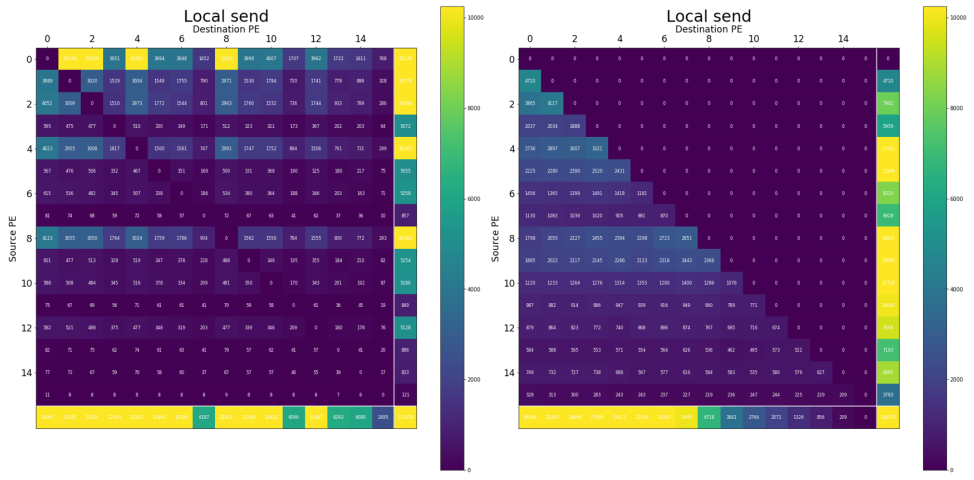

Figure 8: Physical Trace Heatmap for 1 node
(LHS: 1D Cyclic, RHS: 1D Range)

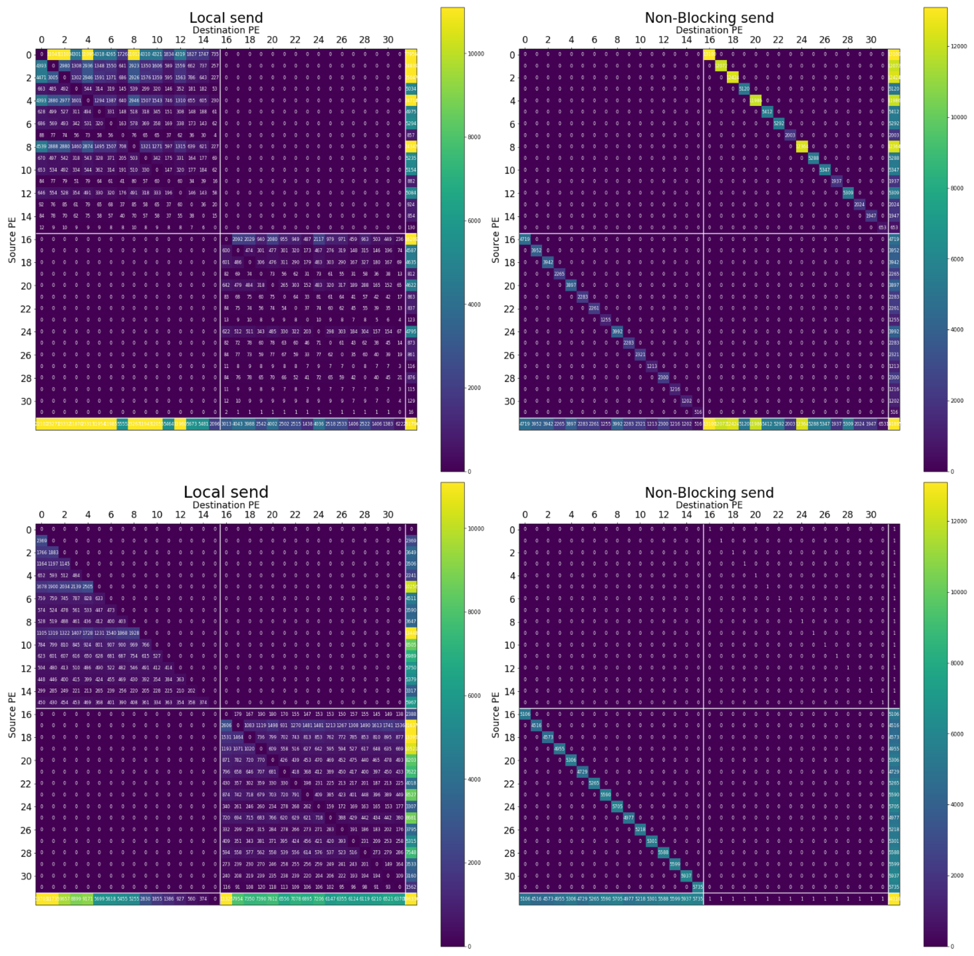

Figure 9: Physical Trace Heatmap for 2 nodes
(UP: 1D Cyclic, BOTTOM: 1D Range)

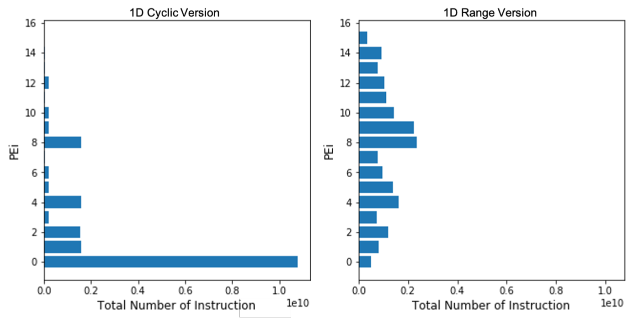

Figure 10: Total Number of Instructions vs PEi for 1 node (LHS: 1D Cyclic [1], RHS: 1D Range)

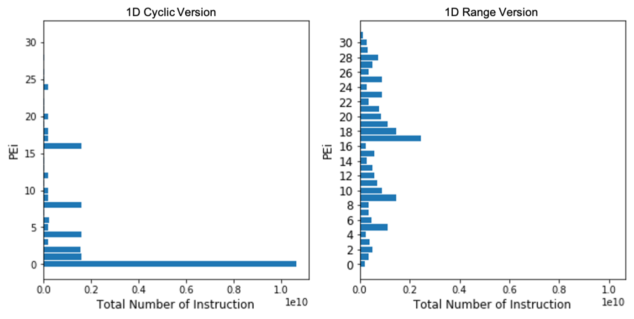

Figure 11: Total Number of Instructions vs PEi for 2 nodes (LHS: 1D Cyclic [1], RHS: 1D Range)

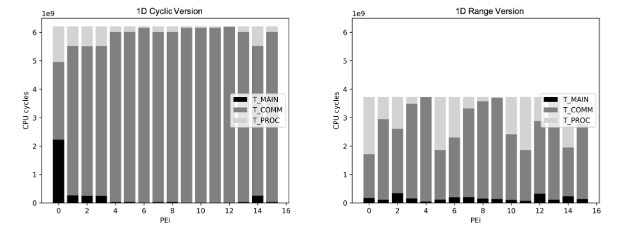

Figure 12: Overall Profiling for 1 node (LHS: 1D Cyclic, RHS: 1D Range)

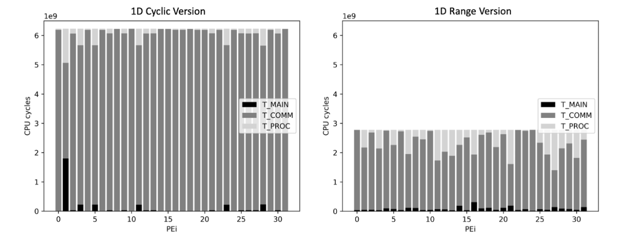

Figure 13: Overall Profiling for 2 nodes (LHS: 1D Cyclic, RHS: 1D Range)

for $\leq 5\%$ of the total time taken for 1D Cyclic and 1D Range over 1&2 nodes. PROC in 1D Cyclic is $\leq 5\%$ whereas for 1D Range is $\sim$20-24% of the total time taken. Therefore, we can infer that although 1D Range is better than 1D Cyclic, it still faces the problem of load-imbalance in recv. Further, time taken in PROC + MAIN accounts for up to 33% of the total. Therefore, the $\sim$2x gain in the total time is primarily from the gain in the time taken in COMM regime. Since changing the distribution improved the performance, ActorProf suggests *experimenting with data-distributions* as an opportunity for improvement.

Conclusion: Triangle counting algorithm is bounded by the time taken by COMM regime. Apart from new distributions, clever techniques could be employed to exploit more overlap between communication and computation in a performant fashion.

E. Overhead of ActorProf Tracing

The FA-BSP model inherently enables the user to send a massive number of messages, potentially bloating the size of logical and physical traces. We discuss the memory problem of logs in Section VI. Additionally, any inferences made from the timestamp information might not be necessarily accurate - Conveyors implements a lazy-send policy whose ordering guarantees are only restricted for a pair of PEs, i.e., if PE0 sends a buffer to PE1 and PE2 sends a buffer to PE1, then PE1 has no way of distinguishing which buffer came first. Furthermore, the inclusion of logical clock setup and its communication in Conveyors/HClib-Actor is costly, as memory-copy/network send overheads are directly proportional to the number of send/recv operations (orders of billion) invoked within the target profiling region.

For Overall profiling, we intentionally used rdtsc so the measurement does not flush the CPU pipeline like rdtscp. While this minimizes profiling overhead, the measured cycles can include cycles taken to execute instructions before or after the measurement. We refrain from using any OS-specific timers and rely on the x86 instruction set instead.

V. Discussion and Related Work

A. Programming Model- & Runtime- Aware Profiling/Tracing

To the best of our knowledge, there are not many profilers designed to capture programming model-specific and runtime-specific information. Chplvis [24] targets Chapel [25] and enables profiling and visualizing tasking and communication information in Chapel programs. Projections [26] is a Java-based performance visualization and analysis tool which targets Charm++ [27] programs. Legion Prof [28], [29] is a task-level profiler that targets the GASNet communication system on Legion [30] programming system. It enables the profiling and visualizing CPU utilization, scheduling, and "happen-before" dependencies between various tasks in Legion programs. Arm Forge [31] profiles MPI and OpenMP programs on conventional HPC architectures - Intel, 64-bit Arm, AMD, OpenPOWER, NVIDIA GPU, and AMD GPU hardware. Similarly, ActorProf targets the **FA-BSP model**, which mainly targets irregular applications, involving heavy context-switching, and aims to bridge the gap between OpenSHMEM-level communication routines invoked from the runtime and fine-grained asynchronous messaging routines invoked from the user.

B. Communication API-level Profiling/Tracing

There are plenty of well-established profilers that can directly profile/trace MPI or OpenSHMEM routines, such as `score-p`, `TAU(`τ`)`, `CrayPat`, `Intel®Vtune™`, and `HPCToolkit`. However, none of them support OpenSHMEM non-blocking routines, which led us to enable Physical Trace in Section III-C. Here are additional details that explain why this is the case:

1) `score-p`: We received an email acknowledgment from the `score-p` team stating that they currently do not support and have no plans to support non-blocking routines.
2) `TAU(`τ`)`: Non-blocking routines are excluded in `exclude_list.openshmem`, which is in `TAU`'s source directory `tau-2.33.2/src/wrappers/shmem/`. APEX [32] by HPX employs `TAU(`τ`)` profiler for their visualization utility.
3) `CrayPat`: We profiled all the bale kernels [22] using `CrayPat` with `"-T"` option enabled, which is supposed to show any OpenSHMEM routines invoked during the execution. However, `shmem_putmem_nbi` does not show up even though Conveyors frequently uses the routine.
4) `Intel®Vtune™`: We consulted the official `Intel®Vtune™` User Guide in the section titled "Contents of Trace Files" [33]. It mentions that it is experimental and utilizes a Fabric Profiler, which only uses `shmem_put` calls on Page 371.

SKaMPI-OpenSHMEM [34] offers an alternative to measuring non-blocking asynchronous point-to-point communication, but one of the limitations of the model is the measurement of `shmem_quiet` for a fixed buffer size. SKaMPI-OpenSHMEM (Page5) - "*In the preamble of the measurement routine, we measure the time to perform a complete non-blocking communication (including the call to `shmem_quiet`) for the same buffer size.*", which assumes after every fixed count of puts of fixed size, the library should call `shmem_quiet`. However, in Conveyors, the trigger of `shmem_quiet` occurs when the second buffer in the double buffering technique is full for **a particular destination**. But the semantics of `shmem_quiet` dictate that the caller PE has to guarantee the completion of all outstanding `puts` for **all the remote destinations** before the previous call. Therefore, Conveyors does not hold the assumption of the SKaMPI-OpenSHMEM tool. We discuss the challenges and the future work for the precise measurement of `shmem_quiet` in Conveyors.

Another potential approach is the OpenSHMEM Profiling Interface, which is analogous to MPI's `PMPI` profiling interface. We may create a wrapper function for non-blocking routines and collect/record any information related to these routines.

VI. Conclusion and Future Work

This paper designs and implements a profiling and visualization framework called `ActorProf`. The framework enables 1) asynchronous point-to-point message-aware profiling with hardware performance counters, 2) overall performance breakdown that is aware of FA-BSP execution, and 3) visualization of these profiling results. We use an FA-BSP version of distributed triangle counting to demonstrate how `ActorProf` helps the user analyze the application.

In our future work, besides exploring the possibility of enabling FA-BSP-aware profiling in existing profilers, we intend to investigate more efficient profiling and visualization techniques (and their inferences) that can handle large traces of orders of 100GB. Since the FA-BSP model inherently enables the user to send a massive number of messages, managing the size of traces becomes challenging, as explained in [35], and it is a key area for future research. Moreover, a feature where `ActorProf` can concurrently generate the trace graph with the program's execution along with the adoption of `OTF` [36] and `Google Trace Events` [37] format, is currently being investigated.

Furthermore, we have found some intriguing methods for setting the standard for profiling upcoming programming models. One such method is the use of a Declarative Language, which can reduce the overhead caused by a profiler and can be easily integrated with any run-time system written in a procedural language (e.g., C) as pointed out by [38]. Finally, intelligent sampling of traces and identifying hotspots using performance modeling can serve as an alternative approach to interpreting applications compared to `ActorProf`.

Acknowledgment

This research is based upon work supported by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

References

[1] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103–111, aug 1990. [Online]. Available: https://doi.org/10.1145/79173.79181

[2] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, ser. PGAS '10. New York, NY, USA: Association for Computing Machinery, 2010. [Online]. Available: https://doi.org/10.1145/2020373.2020375

[3] M. P. Forum, "Mpi: A message-passing interface standard," USA, Tech. Rep., 1994.

[4] 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.

[5] S. R. Paul, A. Hayashi, K. Chen, Y. Elmougy, and V. Sarkar, "A fine-grained asynchronous bulk synchronous parallelism model for pgas applications," *Journal of Computational Science*, vol. 69, p. 102014, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750323000741

[6] A. Hayashi, J. Yang, and Y. Elmougy, "Hclib-actor documentation," Georgia Institute of Technology, Tech. Rep., 2022. [Online]. Available: https://hclib-actor.com

[7] Y. Elmougy, A. Hayashi, and V. Sarkar, "Asynchronous distributed actor-based approach to jaccard similarity for genome comparisons," in *ISC High Performance 2024 Research Paper Proceedings (39th International Conference)*, 2024, pp. 1–11.

[8] Y. Elmougy, A. Hayashi, and V. Sarkar, "Highly scalable large-scale asynchronous graph processing using actors," in *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, 2023, pp. 242–248.

[9] Y. Elmougy, A. Hayashi, and V. Sarkar, "A Distributed, Asynchronous Algorithm for Large-Scale Internet Network Topology Analysis," in *2024 IEEE/ACM 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*, 2024.

[10] U. o. T. Innovative Computing Laboratory (ICL), *Performance Application Programming Interface (PAPI)*, last Accessed: July 28th, 2024. [Online]. Available: https://icl.utk.edu/papi/

[11] S. Singhal, A. Hayashi, and V. Sarkar, "Bottleneck scenarios in use of the conveyors message aggregation library," in *2024 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 322–324. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ISPASS61541.2024.00045

[12] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 723–732.

[13] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ser. OOPSLA '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 519–538. [Online]. Available: https://doi.org/10.1145/1094811.1094852

[14] G. Agha, *Actors: a model of concurrent computation in distributed systems*. Cambridge, MA, USA: MIT Press, 1986.

[15] S. M. Imam and V. Sarkar, "Selectors: Actors with multiple guarded mailboxes," in *Proceedings of the 4th International Workshop on Programming Based on Actors Agents & Decentralized Control*, ser. AGERE! '14. New York, NY, USA: Association for Computing Machinery, 2014, p. 1–14. [Online]. Available: https://doi.org/10.1145/2687357.2687360

[16] L. Wesolowski, R. Venkataraman, A. Gupta, J.-S. Yeom, K. Bisset, Y. Sun, P. Jetley, T. R. Quinn, and L. V. Kale, "Tram: Optimizing fine-grained communication with topological routing and aggregation of messages," in *2014 43rd International Conference on Parallel Processing*, 2014, pp. 211–220.

[17] B. Priest, T. Steil, G. Sanders, and R. Pearce, "You've got mail (ygm): Building missing asynchronous communication primitives," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2019, pp. 221–230.

[18] A. Mysore, K. Ravichandran, Y. Elmougy, A. Hayashi, and V. Sarkar, "Accelerating actor-based distributed triangle counting," *Supercomputing Conferece*, 2023.

[19] S. P. Singhal, S. Hati, J. Young, V. Sarkar, A. Hayashi, and R. Vuduc, "Asynchronous Distributed-Memory Parallel Algorithms for Influence Maximization," in *37th International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2024.

[20] NERSC, *Perlmutter HPE supercomputer*. [Online]. Available: https://docs.nersc.gov/systems/perlmutter/architecture/

[21] graph500, *graph500*, last Accessed: July 28th, 2024. [Online]. Available: http://www.graph500.org/

[22] "bale/ package," last Accessed: July 28th, 2024. [Online]. Available: https://github.com/jdevinney/bale/blob/master/docs/Bale-StGirons-Final.pdf

[23] G. Gill, R. Dathathri, L. Hoang, and K. Pingali, "A study of partitioning policies for graph analytics on large-scale distributed platforms," *Proc. VLDB Endow.*, vol. 12, no. 4, p. 321–334, dec 2018. [Online]. Available: https://doi.org/10.14778/3297753.3297754

[24] P. A. Nelson and G. Titus, "Chplvis: A communication and task visualization tool for chapel," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016, pp. 1578–1585.

[25] B. L. Chamberlain, "Chapel (cray inc. HPCS language)," in *Encyclopedia of Parallel Computing*, 2011, pp. 249–256.

[26] L. Kalé and A. Sinha, "Projections: A preliminary performance tool for charm," in *Parallel Systems Fair, International Parallel Processing Symposium*, Newport Beach, CA, April 1993, pp. 108–114.

[27] L. V. Kalé, *Charm++*. Boston, MA: Springer US, 2011, pp. 256–264. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_242

[28] Stanford, N. A. Laboratory, L. A. N. Laboratory, and NVIDIA, *Legion: Performance Profiling and Tuning*. [Online]. Available: https://legion.stanford.edu/profiling/

[29] A. Heirich, E. Slaughter, M. Papadakis, W. Lee, T. Biedert, and A. Aiken, "In situ visualization with task-based parallelism," in *Proceedings of the In Situ Infrastructures on Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV'17. New York, NY, USA: Association for Computing Machinery, 2017, p. 17–21. [Online]. Available: https://doi.org/10.1145/3144769.3144771

[30] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Washington, DC, USA: IEEE Computer Society Press, 2012.

[31] Arm, *Introduction to Arm Forge*. [Online]. Available: https://developer.arm.com/documentation/101136/22-1/Arm-Forge/Introduction-to-Arm-Forge

[32] K. A. Huck, A. Porterfield, N. Chaimov, H. Kaiser, A. D. Malony, T. Sterling, and R. Fowler, "An autonomic performance environment for exascale," *Supercomputing Frontiers and Innovations*, vol. 2, no. 3, p. 49–66, Nov. 2015. [Online]. Available: https://superfri.org/index.php/superfri/article/view/64

[33] Intel, *Intel Vtune Code Analysis with Fabric Profiler*, last Accessed: July 28th, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/docs/vtune-profiler/user-guide/2023-0/openshmem-code-analysis-with-fabric-profiler.html

[34] C. Coti and A. D. Malony, "Skampi-openshmem: Measuring openshmem communication routines," 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. 63–80. [Online]. Available: https://doi.org/10.1007/978-3-031-04888-3_4

[35] J. E. McClure, M. A. Berrill, J. F. Prins, and C. T. Miller, "Asynchronous in situ connected-components analysis for complex fluid flows," in *Proceedings of the 2nd Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*, ser. ISAV '16. IEEE Press, 2016, p. 12–17.

[36] A. Knüpfer, R. Brendel, H. Brunst, H. Mix, and W. E. Nagel, "Introducing the open trace format (otf)," in *Computational Science – ICCS 2006*, V. N. Alexandrov, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 526–533.

[37] Google, *Google Trace Event*. [Online]. Available: https://docs.google.com/document/d/1CvAClvFfyA5R-PhYUmn5OOQtYMH4h6I0nSsKchNAySU/preview#heading=h.yr4qxyxotyw

[38] Q. Wu, T. Neuroth, O. Igouchkine, K. Aditya, J. H. Chen, and K.-L. Ma, "Diva: A declarative and reactive language for in situ visualization," in *2020 IEEE 10th Symposium on Large Data Analysis and Visualization (LDAV)*, 2020, pp. 1–11.