Fast to Write. Faster to Run. A Fused Collective for Irregular Distributed-Memory Applications

Souvadra Hati
Georgia Institute of Technology
Atlanta, GA, USA
souvadrahati@gatech.edu

Richard Vuduc
Georgia Institute of Technology
Atlanta, GA, USA
richie@cc.gatech.edu

Abstract

Irregular distributed-memory applications often perform local computation and memory operations around a many-to-many communication exchange. We introduce AllScatterApply (ASAP), a fused collective communication primitive for this pattern. ASAP overlaps local and remote operations at fine granularity, scales to 32,000 CPU cores on NERSC Perlmutter, and delivers up to 74x speedup over point-to-point implementations built on the same one-sided runtime library. We evaluate ASAP on genomic k-mer counting (KC) and connected components (CC). For KC, ASAP achieves 2-5x speedup over bulk-synchronous parallel (BSP) baselines and matches the performance of a hand-tuned asynchronous algorithm. For CC, ASAP is 1.8x faster than both optimized BSP and asynchronous baselines. By avoiding expensive graph partitioning, ASAP also achieves 8.8x faster end-to-end performance than a dedicated asynchronous graph-processing system.

Keywords

Communication Primitive, BSP, FABSP

ACM Reference Format:
Souvadra Hati and Richard Vuduc. 2026. Fast to Write. Faster to Run. A Fused Collective for Irregular Distributed-Memory Applications . In *Proceedings of The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '26).* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Irregular, distributed-memory applications often employ a common computational pattern: local compute and memory operations before and after a many-to-many remote-data exchange. They typically adopt a bulk-synchronous parallel (BSP) strategy [2, 19]: resolve local dependencies first, then invoke an optimized many-to-many collective, such as one provided by Message Passing Interface (MPI) [7]. A natural alternative to BSP collectives is to use one-sided communication with explicit management of fine-grained messages [8, 10, 11, 16, 17] supported in MPI, OpenSHMEM [3], HClibActor [15], and Unified Parallel C (UPC) [5, 21]. However, attaining practical performance with asynchronous libraries can be challenging, with BSP strategies often proving more effective unless significant optimization and tuning are applied [10, 12, 13].

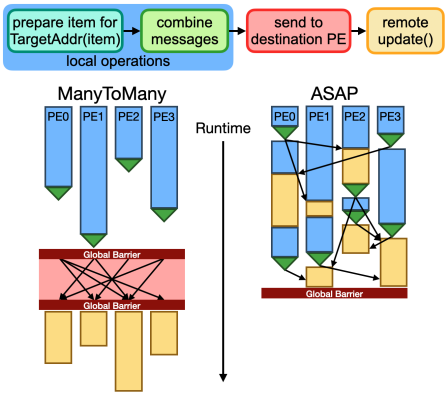


Figure 1: Execution of an irregular application implemented using many-to-many (left) and ASAP (right).

To bridge this performance gap, we introduce AllScatterApply (ASAP), a communication collective that unifies computation and fine-grained communication (Figure 1) while hiding performance-critical implementation details. Users retain the familiar structure of a BSP-style program, introducing synchronization only when algorithmic correctness demands it [14, 15].

We evaluate ASAP on distributed-memory implementations of k-mer counting (KC) and connected components (CC) against state-of-the-art asynchronous and optimized BSP baselines. Some of the highlights from our study are: (1) ASAP scales to 32K CPU cores on NERSC Perlmutter and is 2-74× faster than asynchronous point-to-point implementations built on the same one-sided runtime. (2) For KC, ASAP delivers 2-5× speedup over BSP baseline HySortK [13] and matches the performance hand-tuned asynchronous DAKC [10]. (3) For CC, it is 1.8× faster than the optimized BSP baseline AUXCC and the bulk-asynchronous `GluonAsync`. More importantly, ASAP achieves this performance without expensive graph-partitioning step, making it 8.8× faster end-to-end than a dedicated asynchronous graph-processing system.

2 The ASAP Abstraction

The ASAP abstraction targets a common single program multiple data (SPMD) irregular-computation pattern, illustrated in Figure 2. Each process owns part of a distributed data and, for every local input item, visits a data-dependent set of neighbors and applies an update to their states. A level-synchronous breadth-first search (BFS) step is one such example: frontier vertices are the input, edges define neighbors, and the update sets vertex levels. The same pattern appears in histograms, hash tables, genome assembly, PageRank,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SC '26, Chicago, IL, USA
© 2026 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Algorithm 1: BSP Algorithm using ManyToMany

```
def IrregularApp_ManyToMany(Input, Output):
  SendBuf = []
  for item in Input:
    for dst in TargetAddr(item):
      SendBuf.add({item, dst})
    if (length(SendBuf) >= Batch):
      Exchange(Output, SendBuf)
  Exchange(Output, SendBuf) # Last batch, if any

  def Exchange(Output, SendBuf):
    Compact(SendBuf)
    RecvBuf = ManyToMany(SendBuf) # Can be nonblocking
    for r in RecvBuf:
      Apply(Output[r.dst], r)
    Empty(SendBuf)
```

Algorithm 2: FABSP Algorithm using ASAP

```
def IrregularApp_ASAP(Input, Output):
  CommObj = ASAP(Output, Merge(), Receive(), OwnerPE())
  for item in Input:
    for dst in TargetAddr(item):
      CommObj.Push({item, dst})
  CommObj.Done()
  GLOBAL BARRIER

def Merge(A, B):
  if A.dst == B.dst:
    # User defines how ASAP merges two packets

def Receive(Output, A):
  Apply(Output[A.dst], A)
```

Figure 2: Pseudocode comparing the implementations of the same irregular application using many-to-many and ASAP.

and influence maximization [6, 8, 9, 15, 17]. In the BSP implementation on the left of Figure 2, each process performs local updates and buffers remote ones in `SendBuf`. An `Exchange` may compact this buffer before a many-to-many collective, after which received updates are applied locally. On P processors, each exchange requires $O(\log P)$ or $O(P)$ synchronization steps. Batching limits buffer growth and amortizes communication overhead, but offers only coarse-grained overlap and favors large messages.

The ASAP implementation on the right applies the Fine-Grained Asynchronous Bulk Synchronous (FABSP) model [14, 15] to overlap computation and communication at much finer granularity. The user creates an ASAP communication object, `CommObj`, with three application-specific functions: `Receive` is invoked when a remote message arrives, `Merge` combines similar messages in an application-dependent manner, and `OwnerPE` identifies the destination processor. Instead of buffering remote operations explicitly, the application immediately `Pushes` them to `CommObj`. It is ASAP's job to optimize the compaction, aggregation and routing of those messages. Calling `CommObj.Done()` signals that the process will send no more messages, upon which ASAP ensures that all communication and receive-side operations complete before the next global barrier.

3 Experiments And Analysis

We conducted our experiments on Phoenix and Perlmutter. Phoenix nodes have 24-core dual-socket Intel Xeon Gold 6226 CPUs, 192 GB memory, and Infiniband 100HDR Network Interface Card (NIC). Its software stack includes OpenMPI/4.1.5, MVAPICH2/2.3.7, OpenSHMEM/4.1.5, and GCC/12.3.0. Perlmutter has dual-socket AMD EPYC 7763 (Milan) CPUs, 512 GB memory, HPE Slingshot 11 NIC. Its software stack includes OpenMPI/5.0.7, Cray-OpenSHMEMX/11.5.7, Cray-PMI/6.1.9, and GCC/13.2.1. We conducted KC experiments on Phoenix and CC experiments on Perlmutter. To keep this summary concise, we omit detailed performance results and refer readers to the accompanying poster for the performance numbers.

3.1 k-mer Counting

We use two open-source baselines for KC. `DAKC` [10] is a fully asynchronous KC algorithm implemented using HClib Actor [14] that uses fine-grained one-sided messages to minimize the number of synchronizations among processors. It also includes low-level tuning not present in our implementation, allowing us to see how much of its performance we can recover. `HySortK` [13] is a BSP baseline using hybrid MPI +OpenMP parallelism with **non-blocking** communication collectives. For `HySortK`, we report the runtime for the best MPI rank and OpenMP thread configuration per datapoint.

3.2 Connected Components

We consider two CC baselines. The first is `GluonAsync` [4], a fully asynchronous graph-processing system. The second, AUXCC, is our optimized MPI baseline. Following the distributed label-propagation approach of Slota et al. [18], AUXCC uses an auxiliary data structure to bound worst-case interprocessor communication volume per iteration. To reduce message volume, vertices with remote neighbors, called *interface vertices*, send their information once to each destination processor. Before computation, AUXCC constructs and exchanges the interface-vertex send lists and MPI many-to-many offsets, building an auxiliary mapping from received data to local vertex indices. This mapping is reused across iterations, reducing buffering overhead using fixed message volume per iteration.

4 Future Work

For workloads following our target many-to-many pattern, ASAP provides a simple, flexible abstraction that uses one-sided communication orchestration to achieve aggressive overlap of computation and communication. Building on this foundation, our future work will (1) evaluate ASAP on more workloads, (2) implement ASAP over runtime libraries beyond HClib Actor [15], and (3) combine explicit multithreading alongside asynchronous active messages [20].

5 AI Acknowledgment

Claude [1] was used solely for minor language polishing, including grammatical corrections and rephrasing. The authors edited all AI-assisted text, as well as all scientific content, results, analysis, and references, and assume full responsibility for this paper's contents.

6 Acknowledgment

This material is based in part upon work supported by the U.S. National Science Foundation (NSF) Award Number 2316177. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF.

References

[1] Anthropic. 2026. The Claude 4 Model Family. https://www.anthropic.com/claude. Large language model, accessed 2026-04-07.

[2] Rob H. Bisseling. 2020. *Parallel scientific computation: A structured approach using BSP* (2nd ed. ed.). Oxford University Press. https://global.oup.com/academic/product/parallel-scientific-computation-9780198788355?lang=en&cc=us

[3] Barbara Chapman, Tony Curtis, Swaroop Pophale, Stephen Poole, Jeffc Kuehn, Chuck Koelbel, and Lauren Smith. 2010. Introducing OpenSHMEM: SHMEM for the PGAS Community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (New York, New York, USA) *(PGAS '10)*. Association for Computing Machinery, New York, NY, USA, Article 2, 3 pages. https://doi.org/10.1145/2020373.2020375

[4] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Vishwesh Jatala, V. Krishna Nandivada, Marc Snir, and Keshav Pingali. 2019. Gluon-Async: A Bulk-Asynchronous System for Distributed and Heterogeneous Graph Analytics. In *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques (PACT 2019) (PACT '19)*. IEEE. https://doi.org/10.1109/PACT.2019.00010

[5] Tarek El-Ghazawi, William Carlson, Thomas Sterling, and Katherine Yelick. 2005. *UPC: distributed shared memory programming*. John Wiley & Sons. https://doi.org/10.1002/0471478369

[6] Youssef Elmougy, Akihiro Hayashi, and Vivek Sarkar. 2023. Highly Scalable Large-Scale Asynchronous Graph Processing using Actors. In *2023 IEEE/ACM 23rd International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. 242–248. https://doi.org/10.1109/CCGridW59191.2023.00049

[7] Message P Forum. 1994. *MPI: A Message-Passing Interface Standard.* Technical Report. USA. https://link.springer.com/chapter/10.1007/978-3-0348-8534-8_21

[8] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. 2015. HipMer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–11. https://doi.org/10.1145/2807591.2807664

[9] Priyanka Ghosh, Sriram Krishnamoorthy, and Ananth Kalyanaraman. 2019. PaKman: Scalable assembly of large genomes on distributed memory machines. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 578–589. https://doi.org/10.1109/IPDPS.2019.00067

[10] Souvadra Hati, Akihiro Hayashi, and Richard Vuduc. 2025. An Asynchronous Distributed-Memory Parallel Algorithm for k-mer Counting. In *2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, Los Alamitos, CA, USA, 472–483. https://doi.org/10.1109/IPDPS64566.2025.00049

[11] Steven Hofmeyr, Rob Egan, Evangelos Georganas, Alex C Copeland, Robert Riley, Alicia Clum, Emiley Eloe-Fadrosh, Simon Roux, Eugene Goltsman, Aydın Buluç, et al. 2020. Terabase-scale metagenome coassembly with MetaHipMer. *Scientific reports* 10, 1 (2020), 10689. https://doi.org/10.1038/s41598-020-67416-5

[12] Yuxi Hong and Aydın Buluç. 2024. A sparsity-aware distributed-memory algorithm for sparse-sparse matrix multiplication. In *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–14. https://doi.org/10.1109/SC41406.2024.00053

[13] Yifan Li and Giulia Guidi. 2024. High-Performance Sorting-Based K-mer Counting in Distributed Memory with Flexible Hybrid Parallelism. In *Proceedings of the 53rd International Conference on Parallel Processing*. 919–928. https://doi.org/10.1145/3673038.3673072

[14] Sri Raj Paul, Akihiro Hayashi, Kun Chen, Youssef Elmougy, and Vivek Sarkar. 2023. A Fine-grained Asynchronous Bulk Synchronous parallelism model for PGAS applications. *Journal of Computational Science* 69 (2023), 102014. https://doi.org/10.1016/j.jocs.2023.102014

[15] Sri Raj Paul, Akihiro Hayashi, Kun Chen, and Vivek Sarkar. 2022. A Productive and Scalable Actor-Based Programming System for PGAS Applications. In *Computational Science – ICCS 2022: 22nd International Conference, London, UK, June 21–23, 2022, Proceedings, Part I* (London, United Kingdom). Springer-Verlag, Berlin, Heidelberg, 233–247. https://doi.org/10.1007/978-3-031-08751-6_17

[16] Min Si, Antonio J Pena, Jeff Hammond, Pavan Balaji, and Yutaka Ishikawa. 2015. Scaling NWChem with efficient and portable asynchronous communication in MPI RMA. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE, 811–816. https://doi.org/10.1109/CCGrid.2015.48

[17] Shubhendra Pal Singhal, Souvadra Hati, Jeffrey Young, Vivek Sarkar, Akihiro Hayashi, and Richard Vuduc. 2024. Asynchronous Distributed-Memory Parallel Algorithms for Influence Maximization. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis* (Atlanta, GA, USA) *(SC '24)*. IEEE Press, Article 102, 19 pages. https://doi.org/10.1109/SC41406.2024.00108

[18] George M Slota, Sivasankaran Rajamanickam, and Kamesh Madduri. 2016. A case study of complex graph analysis in distributed memory: Implementation and optimization. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 293–302. https://doi.org/10.1109/IPDPS.2016.93

[19] Leslie G. Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (Aug. 1990), 103–111. https://doi.org/10.1145/79173.79181

[20] Jiakun Yan and Marc Snir. 2025. LCI: a lightweight communication interface for efficient asynchronous multithreaded communication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1043–1059. https://doi.org/10.1145/3712285.3759881

[21] Yili Zheng, Amir Kamil, Michael B Driscoll, Hongzhang Shan, and Katherine Yelick. 2014. UPC++: a PGAS extension for C++. In *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 1105–1114. https://doi.org/10.1109/IPDPS.2014.115