SC24-W: Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis | 979-8-3503-5554-3/24/$31.00 ©2024 IEEE | DOI: 10.1109/SCW63240.2024.00097

An Adaptive Asynchronous Approach for the Single-Source Shortest Paths Problem

Ritvik Rao
University of Illinois
Urbana, IL, USA
rsrao2@illinois.edu

Kavitha Chandrasekar
University of Illinois
Urbana, IL, USA
kchndrs2@illinois.edu

Laxmikant Kale
University of Illinois
Urbana, IL, USA
kale@illinois.edu

***Abstract*—Large-scale graphs with billions and trillions of vertices and edges require efficient parallel algorithms for common graph problems, one of which is single-source shortest paths (SSSP). Bulk-synchronous parallel algorithms such as Δ-stepping experience large synchronization costs at the scale of many nodes, so asynchronous approaches are needed for scalability. However, asynchronous approaches are susceptible to wasteful, speculative execution. We introduce ACIC, a highly asynchronous approach modulated by continuous concurrent introspection and adaptation. Using message-driven concurrent reductions and broadcasts, task-based scheduling, and an adaptive aggregation library, we explore techniques such as evolving windows and generation and prioritized flow of optimal updates, or edge relaxations, aimed at reducing speculative loss without constraining parallelism. Our results, while preliminary, demonstrate the promise of these ideas, with the potential to impact a wider class of graph algorithms.**

***Index Terms*—graphs, parallel algorithms, sssp**

I. Introduction

Graphs are used to represent a variety of structures, such as social and road networks [7]. One common graph problem is the single-source shortest paths problem (SSSP), where the distances from a single source vertex to every other vertex in a graph with weighted edges is calculated. Many SSSP algorithms are driven using edge *relaxations*, where an edge is traversed and the known distance of the vertex at the destination of the edge is *updated* if a shorter path from the source to the vertex is found. Parallel SSSP algorithms rely on the idea that multiple edge relaxations can occur simultaneously. However, to achieve a high level of parallelism, multiple speculative edge relaxations occur at once, and since the final distance of a vertex will only be set once, any previous changes to a vertex distance are, strictly speaking, unnecessary.

One canonical parallel SSSP algorithm is Δ-stepping [10], which classifies all tentative vertex distances into buckets of width Δ, and for each bucket, the edges that originate from vertices in that bucket are relaxed in parallel until the bucket is empty. While Δ-stepping reduces speculation, it does so by using multiple costly global synchronizations. The impact of any load imbalances within each bucket is further amplified by these synchronizations.

Asynchronous algorithms eliminate the cost of synchronizations. For example, one asynchronous SSSP algorithm is distributed control [13], where messages are sent across processors simultaneously using a parallel Bellman-Ford [2] method. However, this approach leads to many unnecessary updates because the application has no global view of the distance value distribution of updates, and therefore no control over the flow of messages based on distance value.

A compromise between Δ-stepping and distributed control is the K-level asynchronous algorithm [8], which uses a parameter k that limits the number of edges in any path from the source vertex that are relaxed before a global synchronization. While this approach reduces the number of synchronizations and tunes k to minimize wasted work, load-imbalance issues during super-steps remain, since processors with large numbers of vertices at the same edge depth from the source vertex may bottleneck any synchronization.

In this paper, we introduce ACIC (Asynchronous Continuous Introspection and Control), a novel, fully asynchronous SSSP algorithm that uses continuous reductions across workers, also known as processing elements (PEs), to gather information about the state of the algorithm at the root PE (PE 0), followed by broadcasts to all PEs containing parameters that allow the algorithm to advance as fast as possible. This paper makes the following contributions:

- ACIC, which uses asynchronous reductions to gather a *histogram* of the distribution of distance values of *updates* $u = (v, d)$, or edge relaxations, followed by a broadcast of various *thresholds* that are designed to speed up the processing of *optimal* updates, which contain the true shortest path from the source vertex s to any vertex $v \in V$, while also preventing the creation of *sub-optimal* updates
- An implementation of ACIC using the Charm++ [9] parallel runtime system, which uses message-driven execution to allow for asynchrony and communication-computation overlap
- A comparison of the performance of ACIC with a state-of-the-art Δ-stepping implementation on two classes of graph inputs. We find that on graphs with uniformly randomly assigned edges, ACIC performs 1.36-1.90x faster than Δ-stepping. However, Δ-stepping is 2.5-3.5x faster on scale-free graphs, due to unmitigated load imbalances during the tail of execution.

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00097

II. ACIC Algorithm

In ACIC, a directed, weighted graph is partitioned across an array of PEs. The partitioning is one-dimensional, which means that each PE has a disjoint subset of vertices and the out-edges of those vertices, and only one PE has direct access to any vertex object. ACIC starts by sending an update $(s, 0)$ from the PE containing the start vertex s. An update is a pair $u = (v, d)$, where the first component is a vertex v and the second is a distance from the source. Once a vertex receives an update, if the received distance value d is less than the current distance $dist(v)$, the update is *accepted* and the vertex's distance value is changed to be the received value. If it is not, the update is *rejected* and processing concludes without any further action. An update is considered *processed* when either the update's distance is determined to be larger than the existing vertex distance, or one new update is created for every out-edge of the vertex.

For accepted updates, a *threshold* t_{pq} is used to determine where the update is placed before further processing. The update is either placed in a priority queue pq if $d \leq t_{pq}$, where t_{pq} is the pq threshold, or placed in a pq_hold array if $d > t_{pq}$. The rationale of using t_{pq} is to restrict the processing of updates to a limited, low-cost range of distance values, minimizing speculative loss if low-quality updates are generated. Additionally, while most SSSP implementations place all incoming updates in the priority queue before looking at their distance values, ACIC first does a distance comparison before using pq. This implementation is designed to quickly reject sub-optimal updates and prevent the creation of more sub-optimal updates.

When a PE becomes idle, meaning that it is not actively executing any code, the Charm++ runtime system triggers a task that pulls updates in pq in increasing distance order. Since multiple updates for any given vertex may exist in pq simultaneously, onward updates are only created if $dist(v) == d$, meaning that only the best update for any given vertex in pq is selected. Once an update is pulled, for every out-edge of the updated vertex, a new update is *created*. For every edge (v, w, c), where v is the origin of the edge, w is the destination of the edge, and c is the cost or weight of the edge, an update $(w, dist(v)+c)$ is created, where $dist(v)$ is the newly updated distance value of v.

If every created update was immediately sent to its destination, the network would be swamped with messages and many sub-optimal updates would be processed. To avoid this, ACIC uses another threshold t_{tram} to hold back sub-optimal updates. Given an update distance d, if $d < t_{tram}$, the update is sent immediately. Otherwise, it is placed in a $tram\_hold$. $tram\_hold$ is an array of lists that holds updates that do not meet this threshold, and updates are removed from $tram\_hold$ when the threshold is increased later on during execution. By using t_{tram} to hold back higher distance updates, it is likely that during the time where a higher distance update is held back, an update to that same vertex with a lower distance will be created and sent to its destination. This would lead to the higher distance update being rejected by its destination, reducing the propagation of sub-optimal updates.

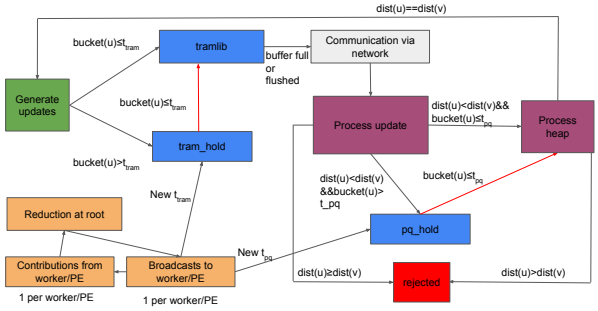

Fig. 1. The lifetime of an update, and how the reduction-broadcast cycle controls how updates are propagated. Green blocks are methods that create updates, purple blocks process updates, blue blocks store updates, orange blocks deal with reductions and broadcasts, and the red *rejected* block represents rejected updates. The red arrows show threshold comparisons that occur when a threshold changes.

To set t_{tram} and t_{pq} while updates are being processed, ACIC uses update histograms. Each PE contains a local histogram, and each histogram is divided into *buckets*, where each bucket represents the number of active updates with distance values between a certain range. Various heuristics may be used to decide the width of each bucket. Currently, ACIC uses the following method to calculate the bucket of an update, given a distance d:

$$bucket(d) = d/\log(|V|)$$

where $|V|$ is the number of vertices in the graph. At update creation time, $bucket(d)$ is incremented, and when the update is considered processed, $bucket(d)$ is decremented. All increments and decrements of buckets happen locally, meaning that the PE that creates an update increments its local bucket, and the PE that processes that same update decrements its local bucket, even if the two PEs are not the same.

Throughout the execution of the algorithm, each PE contributes its local histogram to a sum reduction, producing a global histogram at the root PE (PE 0) that represents the distance distribution of all active updates at the time of the reduction. As shown in fig. 1, reductions occur asynchronously, meaning that after each PE contributes to the reduction, it continues processing and generating updates. Asynchrony allows ACIC to set thresholds without incurring synchronization costs and without stopping the processing of updates. Using this global histogram, the root calculates t_{tram} and t_{pq}, which are set equal to histogram buckets such that the number of active updates in buckets at or beneath the threshold is equal to certain bottom percentiles of all active updates. In the current implementation of ACIC, the percentile use to set t_{pq} is slightly higher than the percentile used to set t_{tram}. This threshold is then communicated to all PEs using a broadcast. During this broadcast, both the $tram\_hold$ and pq_hold are checked for updates that now fall within the new values of

t_{tram} and t_{pq}. Any update in $tram\_hold$ that is within t_{tram} is sent, and any update in pq_hold within t_{pq} is placed in pq. Once a broadcast is complete, another reduction immediately occurs, and cycles of broadcasts and reductions continue until termination.

To send and receive updates, ACIC uses the Tramlib messaging library. Tramlib is a new message aggregation library for Charm++ [6]. Due to the inherently large number of update messages in SSSP, the cumulative effect of message latencies and overheads for each message would cause a large slowdown, reducing scalability. Tramlib is designed for any application with a large number of short messages, with a fine-grained task triggered by each message. It allows for the user to aggregate messages by setting the maximum number of items held in send buffers. Note that the send buffers in Tramlib are distinct from the application-level $tram\_holds$. Once an outgoing buffer reaches this maximum size, Tramlib automatically *flushes* each buffer as an aggregated message to its destination. Additionally, Tramlib supports an explicit flush call that allows the application to manually send all messages even when buffers are not filled up to their maximum size, such as when traversing the "tail" of the graph. We use manual flush calls during broadcasts, guaranteeing that all messages in the Tramlib buffers will eventually be sent.

Execution continues until the system reaches a *quiescent* state. Quiescence is detected by maintaining two counters, one for the number of created updates and one for the number of processed updates. In ACIC, the asynchronous reductions on the histograms also sum up these counter values across all PEs, and the algorithm is considered quiescent when the two counters share an equal value in two consecutive reductions [12]. Upon reaching quiescence, the algorithm terminates.

III. Results and Analysis

A. Implementation

ACIC uses the symmetric multi-processing (SMP) mode in Charm++, where each process has multiple PEs that access a shared memory space. We compare ACIC with a modified Δ-stepping implementation from RIKEN that was designed for running the Graph500 benchmark on Fugaku [11]. The modifications include using a 2D partitioning of the input graph and a hybrid approach that switches from Δ-stepping to Bellman-Ford once a local maxima in the number of newly settled vertices per epoch is reached [4].

B. Datasets

These tests use two classes of automatically-generated graphs as input. The first type is a scale-free graph generated using the recursive matrix (RMAT) method [5]. These graphs, which are designed to emulate most real-world applications, have $|V| = 2^{26}$ vertices and $|V| = 2^{30}$ edges, and demonstrate the power law, where a few vertices have a very high degree and most vertices have a very low degree. The second type is a random, low diameter graph where for each edge, the distance, origin, and destination of the edge are randomly chosen, meaning that the distribution of edge origins and destinations is roughly equal among all vertices. These graphs also have $|V| = 2^{26}$ vertices and $|V| = 2^{30}$ edges, and do not have a power law characteristic. All our input graphs have directed edges, with edge weights being integers and uniformly distributed across the interval $[1, 1000]$, inclusive.

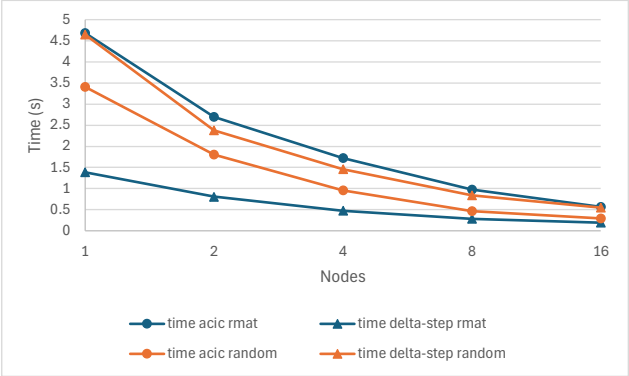

Fig. 2. The execution time of ACIC vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs, scale=26: ACIC is faster on random graphs, but slower on RMAT graphs.

C. Machine and experimental setup

The tests are run on two computing clusters. The first is Delta at the National Center for Supercomputing Applications (NCSA), which has 124 CPU nodes, each of which has a dual AMD 64-core 2.45 GHx Milan processor, 256 GB DDR4-3200 of RAM, and a 800 GB NVM3 solid-state disk. The second is Frontier at the Oak Ridge National Laboratory, which has 9,472 nodes, each with an AMD Epyc 7713 64-core 2 GHz CPU, 128 GB of RAM, and 512 GB of disk storage. For our implementations, we use 48 cores per node on both Delta and Frontier, with 8 processes per node, 6 cores per process for computation (one PE per core), 1 core for a dedicated communication thread per process, and 1 core sacrificed for operating system daemons per process. Each data point represents an average of ten trials at that point. For the randomly generated graphs, different random seeds are used to generate graph structures and edge weights for each trial.

D. ACIC vs. Δ-stepping

We compare ACIC to the modified Δ-stepping algorithm mentioned in the Implementation section. For the random graph, according to fig. 2, our implementation shows a speedup of 1.3x on 1 and 2 nodes, 1.5x faster on 4 nodes, and 1.8x on 8 and 16 nodes. Since the speedup increases as parallelism increases, ACIC shows better weak scaling compared to the Δ-stepping implementation. Additionally, according to fig. 3, ACIC generates 8-16% fewer updates than the Δ-stepping implementation, and according to fig. 4, the number of traversed edges per second (TEPS) of our implementation is 25-63% than Δ-stepping. This means that for random graphs, ACIC reduces unnecessary speculation and processes work faster than a synchronous algorithm. However, for scale-free graphs

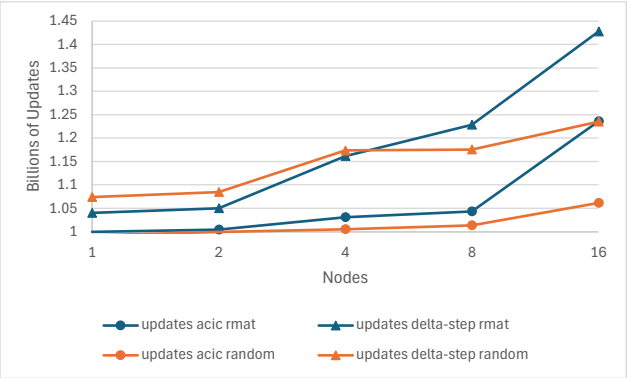

Fig. 3. The number of updates, or edge relaxations, of the Charm++ asynchronous SSSP algorithm vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs: for both type of graphs, Charm++ generates fewer updates. Note that the y-axis starts at 1 billion updates to clarify the differences in update counts between ACIC and Δ-stepping.

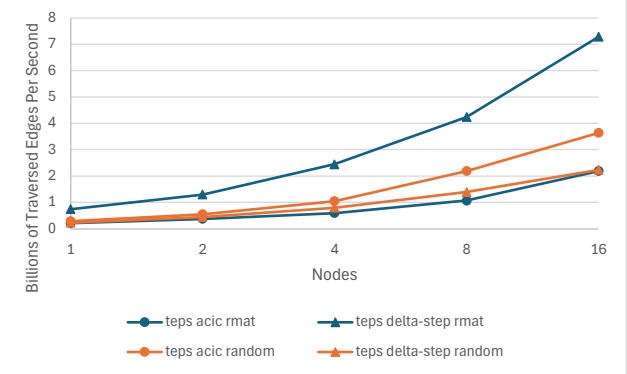

Fig. 4. The traversed edges per second (TEPS) of ACIC vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs: ACIC has higher TEPS on random graphs, but lower on RMAT graphs.

generated with RMAT, ACIC currently underperforms the Δ-stepping alternative. Δ-stepping shows a 2.8 to 3.3x speedup over our implementation, but the magnitude of difference decreases as the number of nodes increases, implying that ACIC could perform better with more nodes and a larger problem size. Compared to ACIC, the TEPS of Δ-stepping is 4x greater at 1 node and 3.5x greater at 16 nodes. As with the random graph, ACIC has 4-18% fewer updates than Δ-stepping.

ACIC is superior on random graphs, since the relatively even distribution of edge degree means that the final distance of each vertex is set much sooner than in Δ-stepping, and the algorithm terminates sooner. However, scale-free graphs have a power-law degree distribution, where a small number of vertices have a high degree. This means that scale-free graphs have a "tail", and during traversal of this tail, concurrency is limited. ACIC does not process this tail efficiently compared to the Δ-stepping approach due to load imbalance issues. The 1-dimensional partitioning used by ACIC creates a greater load imbalance the 2D partitioning used by the RIKEN Δ-stepping algorithm, since vertices with high degree are only accessible on one partition. This constrains the generation of updates from large-degree vertices to whichever PE contains that vertex's data, limiting parallelism. Additionally, if vertices in the tail of the graph are contained on one PE, then those vertices are processed sequentially, increasing runtime if there is a long tail.

IV. Future Work

We plan to explore multiple approaches to improve performance on on scale-free graphs, which represent many real-world large graph datasets. One alternative to our current partitioning method is the 2D partitioning used by the hybrid Δ-stepping algorithm [11], which divides the adjacency matrix of the input graph in two dimensions across the available processors, which helps improve locality. Another alternative is 1.5D partitioning [3], which splits vertices into distributed subgraphs based on degree, ensuring high locality from all processors to vertices with the highest degrees. A more aggressive approach to load balancing is based on the idea of over-decomposition. By dividing the graph into a larger number of partitions than processors, we will be able to enlist the help of other workers within a process, via migrating an entire partition from one processor to another during the tail portion of the execution. In addition to load balancing, this may also help with cache performance by enhancing locality.

We will also generalize ACIC to work on additional graph classes, beyond the scale-free and random graphs tested in this paper. Other types of graphs that may particularly benefit from an asynchronous SSSP approach include high-diameter graphs, which are graphs with a high average path length between vertices, such as the Road graph in the GAP Benchmark Suite [1]. Synchronous implementations may perform poorly on such graphs, since such implementations need more phases, and therefore synchronizations, to traverse paths. Our "controlled asynchrony" approach has a potential to show substantial improvement, since it allows for a finer trade-off between speculative loss and parallelism, without the pitfalls of synchronous phases.

Although SSSP is used as a proving ground for our approach here, we believe our introspective-control approach has a potential for significant benefits for many classes of graph algorithms that require balancing between aggregation. prioritization, speculation and parallelization.

V. Conclusion

We presented ACIC, a new asynchronous algorithm for the SSSP problem that overlaps computation with continuous reductions and broadcasts to monitor the state of the algorithm and control the flow of updates. We analyzed the effects of our techniques on various graph types, and showed that we improve on a state-of-the-art synchronous algorithm on random graphs. Finally, we explored future optimizations to ACIC that will help improve performance on other types of graphs, such as scale-free graphs.

REFERENCES

[1] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[2] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, pages 87–90, 1958.

[3] Huanqi Cao, Yuanwei Wang, Haojie Wang, Heng Lin, Zixuan Ma, Wanwang Yin, and Wenguang Chen. Scaling graph traversal to 281 trillion edges with 40 million cores. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 234–245, 2022.

[4] Venkatesan T Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 28(7):2031–2045, 2016.

[5] Deepayan Chakrabarti, Yiping Zhan, and Christos Faloutsos. R-mat: A recursive model for graph mining. In *Proceedings of the 2004 SIAM International Conference on Data Mining*, 04 2004.

[6] Kavitha Chandrasekar and Laxmikant Kale. Shared memory-aware latency-sensitive message aggregation for fine-grained communication. In *IA3@ SC*, 2024.

[7] Martin Grandjean. A social network analysis of twitter: Mapping the digital humanities community. *Cogent arts & humanities 3, no. 1*, 12 2016.

[8] Harshvardhan, Adam Fidel, Nancy M. Amanto, and Lawrence Rauchwerger. Kla: A new algorithmic paradigm for parallel graph computations. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 27–38, 08 2014.

[9] Laxmikant Kale and Sanjeev Krishnan. Charm++: A portable concurrent object oriented system based on c++. *ACM Sigplan Notes*, 28, 10 1995.

[10] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms 49, no. 1*, pages 114–152, 10 2003.

[11] Daniel Rehfeldt, Katsuki Fujisawa, Thorsten Koch, Masahiro Nakao, and Yuji Shinano. Computing single-source shortest paths on graphs with over 8 trillion edges. In *ZIB Report 22-22*, 11 2022.

[12] Amitabh B. Sinha, L. V. Kale, and B. Ramkumar. A dynamic and adaptive quiescence detection algorithm. Technical Report 93-11, Parallel Programming Laboratory, Department of Computer Science , University of Illinois, Urbana-Champaign, 1993.

[13] Martin Zalewski, Thejaka Amila Kanewala, Jesun Sahariar Firoz, and Andrew Lumsdaine. Distributed control: priority scheduling for single source shortest paths without synchronization. In *IA3@ SC*, pages 17–24, 11 2014.

Appendix: Artifact Description

Artifact Description (AD)

VI. Overview of Artifacts

A_1	https://github.com/charmplusplus/charm
A_2	https://github.com/ritvikrao/charm_graph_code
A_3	https://github.com/farkhor/PaRMAT
A_4	https://github.com/RIKEN-RCCS/Graph500-SSSP

VII. Artifact Identification

A. Computational Artifact A_1

Artifact Setup (incl. Inputs)

Hardware: Our evaluations are run on compute nodes on the Delta supercomputer at NCSA and the Frontier supercomputer at ORNL. Each Delta compute node has a dual AMD 64-core 2.45 GHx Milan processor, 256 GB DDR4-3200 of RAM, and a 800 GB NVM3 solid-state disk. Each Frontier compute node has 9,472 nodes, each with an AMD Epyc 7713 64-core 2 GHz CPU, 128 GB of RAM, and 512 GB of disk storage. Our experiments use between one and 16 nodes on Delta and Frontier.

Software: https://github.com/charmplusplus/charm

Installation and Deployment: In the top-level charm folder, run:

```
./buildold charm++ ofi-linux-x86_64 cxi
gcc smp slurmpmi2cray
--with-production --enable-tracing
-j16
```

B. Computational Artifact A_2

Artifact Setup (incl. Inputs)

Software: https://github.com/ritvikrao/charm_graph_code

Datasets/Inputs: RMAT input graphs are provided by generating artifact A_3.

Installation: In the Makefile, set the CHARM_SMP variable to

```
<charm_install_directory>/ofi-linux-x86_64
-cxi-slurmpmi2cray-smp-gcc/bin/charmc
```

Then, in the charm_graph_code directory, run

```
make weighted_htram_smp
```

Artifact Execution

To generate a graph with uniformly randomly assigned edges and then run ACIC, do:

```
./weighted_htram_smp +p<num PEs>
<num_vertices> <num_edges> <random_seed>
<start_vertex> 1 +ppn<PEs per process>
+setcpuaffinity
```

To read in a edge list in CSV format (used for other types of graphs, including RMAT), run the following command:

```
./weighted_htram_smp +p<num PEs>
<num_vertices> <path_to_edgelist>
<random_seed> <start_vertex>
0 +ppn<PEs per process> +setcpuaffinity
```

The edge list must be sorted in ascending order by vertex number of the origin of each edge. On Frontier, use weighted_htram_smp_frontier.sh to submit jobs. On Delta, use weighted_htram_smp.sh.

C. Computational Artifact A_3

Artifact Setup (incl. Inputs)

Software: https://github.com/farkhor/PaRMAT

Installation: From the Release folder, run make.

Artifact Execution

In the release folder, generate an edge list with the following command:

```
./PaRMAT -nVertices <vertices>
-nEdges <edges> -threads <cores>
-sorted -noEdgeToSelf -noDuplicateEdges
```

This will generate output to a file called out.txt. Then in the rmat_preprocess.py script in charm_graph_code, change line 4 to point to out.txt, and change line 5 to define an output csv file. Then run:

```
python3 rmat_preprocess.py
```

D. Computational Artifact A_4

Artifact Setup (incl. Inputs)

Software: https://github.com/RIKEN-RCCS/Graph500-SSSP

Installation: Before installing, in src/utils/parameters.h, line 170, change the 16 to 10. Then, in apps/main.cc, line 131, change the line to:

```
#if 0
```

Then, based on graph type, in src/sssp/benchmark_helper.hpp, change lines 310 and 311 to:

```
<Rmat/Random>
GraphGenerator<typename
EdgeList::edge_type,
5700, 1900> generator(scale,
edge_factor, 1000,
PRM::USERSEED1, PRM::USERSEED2,
InitialEdgeType::NONE);
```

Then install cmake, then run the following commands:

```
mkdir build && cd build && cmake ..
make
```

Artifact Execution

```
export MPI_NUM_NODE=<num_cores>
srun -n <nodes> -N <processes_per_node>
./bin/sssp-parallel <scale>
```