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 perform well on moderately-sized workloads, but the synchronization costs at a multi-node scale are large, so asynchronous approaches are needed for very large-scale graphs. However, asynchronous approaches to date are likely to suffer from large amounts of wasted, 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 networks and road networks [8]. 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. Sequential approaches to SSSP include Dijkstra's algorithm [6], which uses a min-priority queue to traverse a graph in as few steps as possible, and the Bellman-Ford algorithm, which iteratively relaxes all edges in a graph and updates distances in $|V|-1$ loops [2]. Algorithms such as Dijkstra's are known as *label-setting*, where once a vertex is "visited" by the algorithm, the set distance of the vertex is considered final, while algorithms such as Bellman-Ford are *label-correcting*, where vertices can be visited and updated multiple times during the algorithm.

As graphs have grown in size, parallel algorithms have been developed to solve the SSSP problem faster. Parallel SSSP algorithms rely on the idea that multiple edge relaxations can occur simultaneously. Additionally, most parallel approaches are label-correcting, which helps maximize parallelism. However, to achieve this parallelism, multiple speculative edge relaxations occur at once, and many vertices have their distances changed repeatedly. Since the final distance of a vertex will only be set once, any previous changes to a vertex distance are unnecessary and such updates should be avoided if possible.

One canonical parallel SSSP algorithm is Δ-stepping [11], 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. Δ is a tunable parameter; a lower value of Δ reduces parallelism but minimizes the number of vertex distance changes, while a higher value of Δ increases parallelism while also increasing the number of speculative updates. While Δ-stepping minimizes unnecessary updates, it does so by splitting phases of the algorithm with multiple global synchronizations. If a load imbalance exists during a phase of Δ-stepping, many processors may sit idle while waiting for one processor to reach the synchronization barrier. Modern graph datasets often have billions and trillions of vertices and edges, and require multiple nodes to process. Accordingly, synchronization costs due to load imbalance are very large across nodes.

To eliminate the cost of synchronization, asynchronous algorithms are ideal for large-scale distributed memory graphs. One asynchronous SSSP algorithm is distributed control [14], where messages containing a vertex number and distance value are sent between processors, and the distance of the vertex is updated if the distance value in the incoming message is smaller than the current vertex distance. The algorithm terminates when no messages are left to be processed on any processor. While distributed control solves the problem of synchronization, this approach leads to many unnecessary updates because the application has no global view of the distance value distribution of updates (only a view of the rate of creation and processing of updates), which would help reduce the propagation of sub-optimal messages.

A compromise between Δ-stepping and distributed control is the K-level asynchronous algorithm [9], which contains a parameter k that limits the number of edges in any path from the source vertex that are relaxed before a global synchronization. The phases of the KLA algorithm are called super-steps, and vertices that can't be reached within the next k iterations of the current super-step are deferred to the next step. At each synchronization, k is either doubled, halved, or kept constant based on the number of vertices whose distances were changed during the last super-step. While this

approach reduces the number of synchronizations and tunes k to minimize wasted work, load-imbalance issues during supersteps 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 distance of the shortest path from the source vertex s to any vertex $v \in V$, while also preventing the creation of *sub-optimal* updates that do not contribute to the final result of the SSSP algorithm.
- Novel asynchrony-focused optimizations, such as the use of a min-priority queue on each PE to only hold updates that improve the distance of a vertex, to further reduce the number of generated updates
- An implementation of ACIC using the Charm++ [10] parallel runtime system, which uses message-driven execution to allow for asynchrony and communication-computation overlap
- An evaluation of ACIC on two types of input graphs, and a comparison of the performance of ACIC with a state-of-the-art Δ-stepping implementation. 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 graphs generated using the recursive matrix (RMAT) method.

II. ACIC Algorithm

A. Baseline algorithm

In our algorithm, a directed, weighted graph is partitioned across an array of processing elements. Each vertex object contains a tentative distance from the source vertex (initialized to 0 on the source vertex s and ∞ on all other vertices) and a list of edges, each of which has a destination and a weight. The partitioning is one-dimensional, which means that each PE has a fraction of vertices and the out-edges of those vertices, only one copy of a vertex object exists on one PE, and no other PE has direct access to the information for that vertex. The baseline label-correcting asynchronous algorithm starts by sending updates $u = (v, d)$ from the PE containing the partition that contains vertex s. An update is a pair, where the first argument is a vertex v and the second argument is a distance from the source. Updates are equivalent to edge relaxations in sequential SSSP algorithms such as Bellman-Ford. Once a vertex receives an update, it compares the received distance value to its current distance value. If it is less than the current distance, the distance value of the update is set as the new distance, and for every out-edge of the vertex, a new update is generated. If it is not, the processing of the update concludes without any further action.

An update $u = (v, d)$ is considered *created* when the update object is created, and *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 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. This update is then sent from the PE containing the partition with v to the PE containing the partition with w. Execution continues until the system reaches a *quiescent* state, meaning that the number of updates created but not yet processed is equal to zero on every PE.

Algorithm 1: Reductions at the root

```
1  Function ReduceHistogram(histogram, bucket_count) :
2  {
3  if updates_created == updates_processed then
4  |  Terminate;
5  end
6  else
7  |  histo_sum ← 0;
8  |  for i ← 0 to bucket_count − 1 do
9  |  |  histo_sum ← histo_sum + histogram[i];
10 |  end
11 |  if histo_sum <= |PE| * 100 then
12 |  |  t_tram ← bucket(1); t_pq ← bucket(1);
13 |  end
14 |  else
15 |  |  t_tram ← bucket(p_tram); t_pq ← bucket(p_pq);
16 |  end
17 |  broadcast(t_tram,t_pq);
18 end
19 }
20 Function bucket(p, histogram,
21 bucket_count, histo_sum) :
22 {
23 current_sum ← 0;
24 for i ← 0 to bucket_count − 1 do
25 |  current_sum ← current_sum + histogram[i];
26 |  if (current_sum >= (p/100) * histo_sum) then
27 |  |  return i;
28 |  end
29 end
30 }
```

B. Update histogram

The primary flaw of the baseline asynchronous approach is that numerous updates will be generated that will not contain the final distance value of the vertex targeted by the update. Such updates are considered *wasted*. In a hypothetically work-minimal SSSP algorithm, each vertex will receive an update exactly once, and that update will contain the length of the shortest path from the source to that vertex, meaning there are no wasted updates. This is impossible, but since wasted updates increase communication overheads without meaningfully advancing the algorithm, an SSSP algorithm should strive to reduce the total number of updates as much as possible. Algorithms such as Δ-stepping rely on phases, split by synchronizations, to determine which edges from which vertices can be relaxed. While this reduces wastage, it leads to increased synchronization costs.

To reduce the number of wasted updates while also maintaining asynchrony, 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 (created but not yet processed) with distance values between a certain range. The range of each bucket is equal in length, and is currently calculated using the following formula:

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

where $|V|$ is the number of vertices in the graph.

When an update is created, the value of the histogram bucket corresponding to the distance of the update is incremented, and when the processing of that update is completed, the histogram bucket 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.

During 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. Using this global histogram, the root calculates the bucket *threshold*. A threshold is a histogram bucket such that the number of updates whose distance is at or beneath the threshold is equal to a certain bottom percentile of all active updates. This threshold is then communicated to all PEs using a broadcast. The purpose of the threshold is to provide an upper bound on the distance of newly created updates that may be sent to their destinations. Updates within the threshold are sent immediately, while updates above the threshold are held back until the threshold is increased later in the program. By holding back higher distance updates, it reduces the probability that those updates will wastefully change the tentative distances of their target vertices, which would create even more updates. During the time where a higher distance update is held back, it is likely that a update to that same vertex with a lower distance will be created and sent to its destination, which would lead to the higher distance update being rejected by its destination and preventing the propagation of sub-optimal updates.

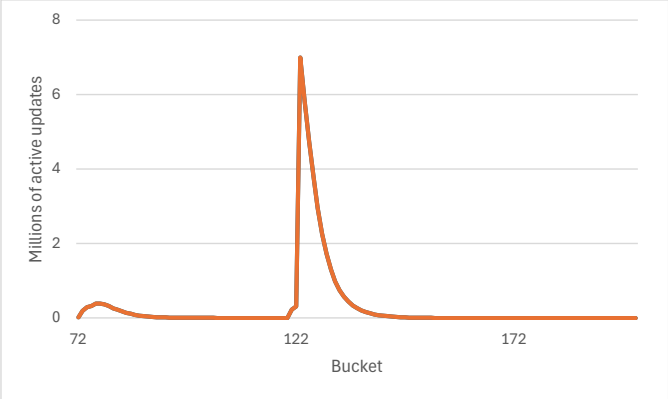

Fig. 1. An example of an aggregated histogram at the root during the middle of a one-node RMAT graph run where $p_{tram} = 0.1$: there are 512 buckets. The lowest bucket number with remaining updates is 72, since all updates with lower distances have been processed already. The very large peak represents updates above t_{tram}, meaning they are in tram holds, waiting for t_{tram} to increase so that they can be sent to their destinations via tramlib. The smaller peak is the result of updates in various priority queues and pq_holds. There is a flat region between the peaks because those updates are the higher-distance updates that were within t_{tram}. These updates are more likely than the lowest distance updates to be rejected once received at their destinations, so there are very few surviving updates between the peaks. In short, this histogram shows that updates below the threshold tend to be processed quickly, and updates above the thresholds are processed more slowly, which is the intent of the threshold.

C. Creation and processing of updates

When an update $u = (v, d)$ is created, the PE that created the update must send it to its destination PE. The destination PE is the PE that has the graph partition that contains vertex v. To limit the proliferation of sub-optimal updates, a threshold is maintained to determine which updates are sent instantly and which updates are held back in a structure known as a $tram\_hold$. The threshold refers to a histogram bucket, and to find out whether an update falls within the threshold t, the histogram bucket for that update is calculated ($bucket(d)$). If $bucket(d) <= t$, the message is sent to its destination. A send from the perspective of the application means that the update is inserted into the tramlib library, which holds the update into an outgoing buffer until it is sent to the destination PE at a later time. If $bucket(d) > t$, update u is held in the $tram\_hold$ until t is increased such that $bucket(d) <= t$. At the broadcast occurring at the end of every reduction, any update whose bucket is within the new threshold is placed into tramlib. $tram\_hold$ is an array of lists that holds updates based on their bucket value, so when updates are moved from $tram\_hold$ to tramlib, the updates are placed in increasing bucket value, ensuring that updates with the lowest distances are sent earlier. Eventually, the threshold will be raised to be equal to the last bucket, which allows all updates to be sent, and therefore allowing the algorithm to terminate.

When update u arrives at its destination PE, the distance of vertex v is compared to d. If $dist(v) < d$, then $dist(v)$ is set to d immediately. If $dist(v) > d$, the update is rejected and considered processed. For accepted updates, the update is either placed in a priority queue pq if $bucket(d) <= t_{pq}$,

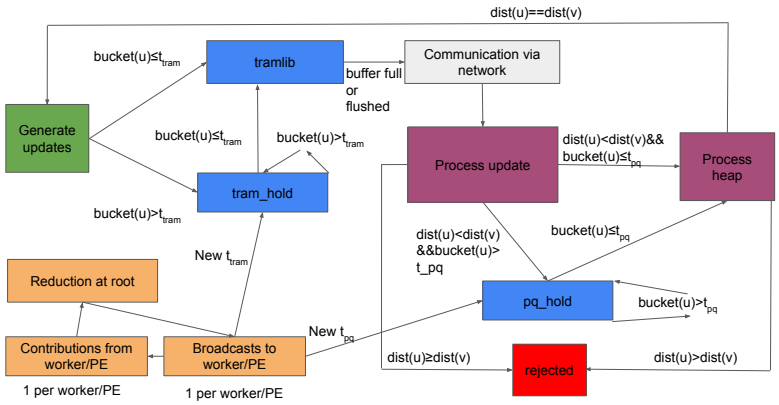

Fig. 2. The lifetime of an update, and how the reduction-broadcast cycle controls how updates are propagated. Green blocks are methods that create updates, purple blocks process updates, blue blocks store updates, orange blocks deal with reductions and broadcasts, and the red *rejected* block represents rejected updates.

where t_{pq} is the pq threshold, or placed in a pq_hold if $bucket(d) > t_{pq}$. Updates move from pq_hold to pq during the broadcast after every reduction. The rationale of using pq_hold is to further delay the processing of sub-optimal updates. If all updates were immediately placed in pq, then a sub-optimal update would be pulled from the queue before a newer, better update is created and delivered.

When a PE becomes idle, meaning that it is not actively executing any code, the Charm++ runtime system triggers a method that pulls updates in pq in increasing distance order. Onward updates are only created if $dist(v) == d$, meaning that this update was not replaced with a better one before it was popped from pq. This usage of the priority queue contrasts with most SSSP implementations, which place all incoming updates in the priority queue. By only placing updates that improve the distance of vertices in pq, and by reducing $dist(v)$ as early as possible, we can quickly reject any update that would not improve the distance of a vertex. We also reduce the creation of updates by waiting for better updates to arrive, and only creating updates based on the best known distance value.

D. Tramlib messaging library

The tramlib messaging library is a new messaging library for Charm++. Due to the inherently large number of update messages in SSSP, if each update was communicated separately as its own message, the cumulative effect of message latencies and overheads for each message would cause a large slowdown, reducing scalability. Since the messaging cost of an application is much more dependent on the number of messages than the size of each message, it is generally optimal to hold outgoing messages in a buffer and only send the message once the number of messages in the buffer reaches a certain value. Tramlib allows for the user to determine the maximum size of outgoing buffers and adjust them based on the number of PEs and the size of the input (such as the number of edges in the context of SSSP). Once an outgoing buffer reaches this maximum size, tramlib automatically *flushes*, or sends, each message to its destination. Additionally, tramlib supports an explicit flush call that allows the application to manually send any messages in its buffers, even if they are not full. This is required for the SSSP application to advance, especially during the last portion of the algorithm, when the "tail" portion of the graph is being traversed. In this "tail" portion, almost all vertices are finalized, except for a small number of vertices that are very far away from the source. Since there is very little concurrency in the tail, message buffers do not fill up, which requires manual flushing to advance the algorithm. Our solution is to add a manual flush call during the broadcast after every reduction, which guarantees that all messages in the tramlib buffers will eventually be sent.

Another feature provided by tramlib is the organization of sending buffers. Tramlib provides the option of either maintaining one set of buffers for each PE or worker (W), or one set of buffers for each process (P). In the latter case, since buffers can be added to by multiple PEs, atomic operations are used to prevent race conditions during writes to the buffers. A given set of buffers includes either one buffer per destination PE, or one buffer per destination process. The buffer size parameter refers to the size of each one of these buffers, so the optimal buffer size for per-PE buffers is lower than the optimal for per-process buffers, assuming the number of PEs and processes is equal. We use two-letter designations to decide which type of aggregation is used. For example, PW aggregation means there is one set of buffers per process, and each set has one buffer per destination PE. Tramlib currently offers PP, WP, and WW aggregation. We found that for SSSP, WP aggregation performs best, so all experiments use WP aggregation.

E. Algorithm termination

In a bulk-synchronous SSSP algorithm, termination conditions can be checked during each synchronization. For example, in the baseline Δ-stepping algorithm, when no vertices remain in any bucket, the program terminates. However, in an asynchronous approach, termination must be detected during execution. The most common approach is to detect quiescence, which means checking that there are no remaining active messages or updates. Each PE maintains counts of locally created and processed messages, and these counts are combined across all PEs using a reduction. When these counts match, then the optimal distance to all vertices from the source is known, and the algorithm is complete. This quiescence detection is generally carried out at the level of the runtime system, and Charm++ does implement a quiescence detection algorithm [13]. However, tramlib messages (updates in the SSSP algorithm) are not accounted for by the runtime-level counters used by Charm++. Therefore, we take advantage of the fact that we already implement reductions for the histograms to detect quiescence. Each PE has two counters. The *updates created locally* counter is incremented when an update is created, and *updates processed locally* is incremented when either an update is rejected or all onward updates are created. These counters are reduced along with the histogram, and during every reduction, they are compared. If they are equal, and their values haven't changed since the last reduction, the algorithm has terminated. We require that the equality condition is met in two consecutive reductions, to account for a race condition where the counters may be equal but unprocessed messages still exist.

One other type of termination condition we experimented with is based on the idea that if the distance of a vertex is smaller than the smallest distance value of any active update, the distance of that vertex will not reduce any further, and can be considered final. Each PE maintains a local count of finalized vertices using this metric, and this count is reduced. If all vertices are finalized, we can ignore any remaining active update and end the algorithm immediately. All our input graphs have non-negative edge weights, allowing this condition to hold true. However, when implementing this, we noticed that for a given source vertex, there would be some vertices that are unreachable from that vertex, meaning that we could not know how many vertices needed to be finalized for the algorithm to terminate ahead of time. Therefore, we ultimately abandoned this termination condition and continued with detecting quiescence.

III. ACIC Parameters

One of the goals of ACIC is to create several tunable parameters that allow the SSSP algorithm to execute as fast as possible for any given input.

a) Tram threshold: The tram threshold (t_{tram}) determines whether an update is immediately placed in a send buffer in tramlib, or is placed in an application-level hold buffer, where it stays until a reduction cycle causes the tram threshold to increase, after which the update is placed in tram. A lower tram threshold slows the propagation of sub-optimal updates, and therefore the generation of wasted work, at the expense of parallelism. A higher tram threshold increases parallelism but results in more wasted work.

To find this threshold during each update, a simple metric is used: If the number of active updates is low, the amount of parallelism is limited, so t_{tram} is set as the highest histogram bucket, meaning that all updates are directly sent to tramlib. Our definition of low is 100 times the number of PEs ($100 * |PE|$), meaning that there are no more than 100 active updates on each PE. If the number of active updates $u_{active} > 100*|PE|$, then each histogram bucket is iteratively added at the root, starting from the smallest bucket with at least one update, until $p_{tram}\%$ of all updates in the histogram have been counted, where p_{tram} is a percentile provided by the user at runtime. The threshold is then set to the current bucket in this loop and then broadcasted to all PEs.

b) PQ threshold: The priority queue threshold (t_{pq}) applies to updates that are added to the priority queue. Below the threshold, they are immediately added to the priority queue, and above it, they are place in a *pq_hold*, until a reduction increases the threshold. The trade-offs for this threshold are the same for the tram threshold, where a lower threshold reduces wasted work and a higher threshold increases parallelism.

t_{pq} is determined in a similar way to t_{tram}. If $u_{active} < 100*|PE|$, then $t_{tram} = num\_buckets - 1$. Otherwise, t_{tram} is set to the bucket where p_{pq} is reached, where p_{pq} is another percentile provided by the user at runtime.

c) Tramlib buffer size: The tramlib buffer size controls the level of message aggregation. A larger buffer reduces the number of messages sent during the application. However, if set too large, messages can spend too much time stuck in a message buffer before being sent and processed. A smaller buffer helps messages get to their destinations faster, but increases network contention and overhead.

IV. Results and Analysis

A. Implementation

ACIC uses the symmetric multi-processing (SMP) mode in Charm++, where there are multiple processes, and 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 [12]. The modifications implemented 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 types of automatically-generated graphs as input. The first type is a scale-free graph generated using the recursive matrix (RMAT) method [5]. These graphs have $|V| = 2^{26}$ vertices and $|V| = 2^{30}$ edges, and demonstrate a power law common in many real-world graphs, where a few vertices have a very high degree and most vertices have a very

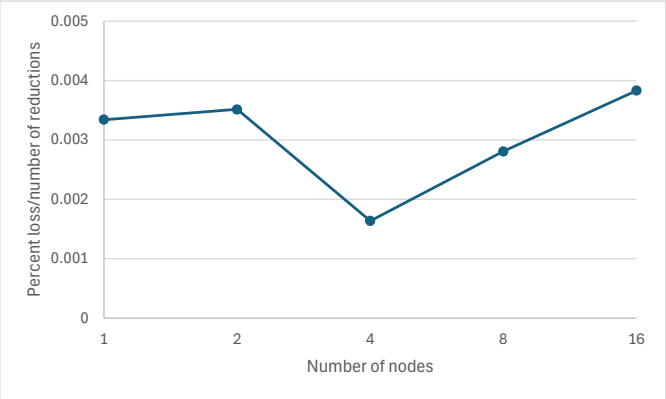

Fig. 3. This compares the percentage reduction in the number of work methods executed at different levels of parallelism, normalized by the number of reductions that occur. The overhead of each reduction and broadcast cycle is very small, and has little effect on the work completed by a PE.

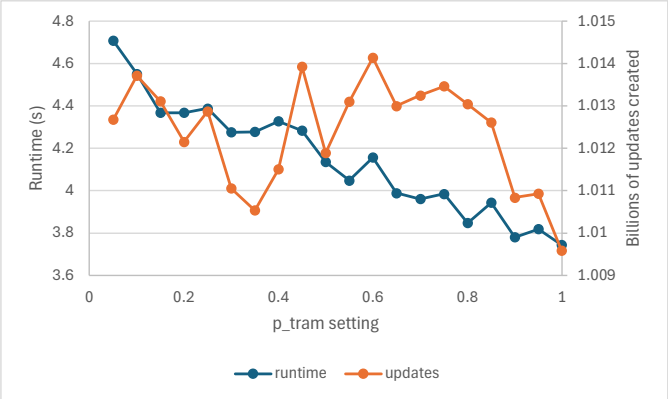

Fig. 4. The effect of the tram percentile on the runtime of ACIC on a graph with randomly assigned edges, run on one node with 48 PEs: the optimal value of p_{tram} is 0.999, meaning all created updates are immediately sent via tramlib.

low degree. The second type is a random, low diameter graph where for each edge, the distance, origin, and destination of the edge is 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.

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 here 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. Reduction overhead

On Frontier, we wrote a standalone Charm++ program to calculate the overhead of our concurrent reductions. Over a period of 5 seconds, each PE repeatedly executes 10 microsecond-long methods to simulate the processing and generation of updates. We then count the number of these methods executed per second across all PEs with and without a concurrent cycle of broadcasts and reductions, and see how many fewer methods are executed when reductions are turned on, normalized by the number of reductions that actually take place. When reductions occur, each PE must pause the execution of work to contribute to reductions and receive

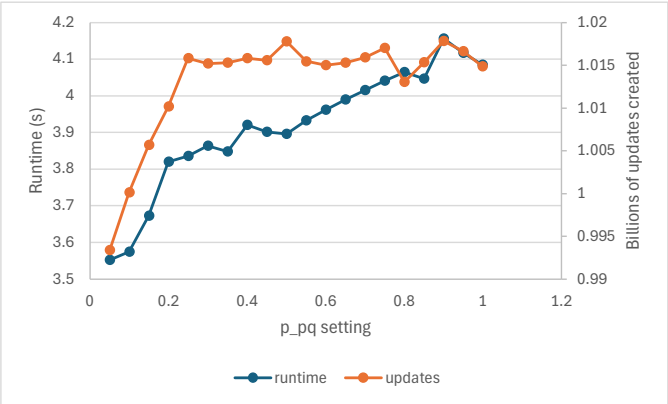

Fig. 5. The effect of the pq percentile on the runtime of our SSSP approach on a graph with randomly assigned edges, run on one node with 48 PEs: The optimal value of p_{pq} is 0.05, meaning only the updates with distances under the 5th percentile should be placed onto pq, while the rest should be placed in pq_hold.

broadcasts, which will cause a reduction in the amount of work completed. However, our tests show that each reduction per second only results in a 0.0015-0.0035% loss in the amount of work done, so if 300 reductions per second happen in a program, there is only a 0.45%-1.05% loss in work each second. This shows that reductions are a viable strategy to create an overview of the SSSP algorithm asynchronously during reductions.

E. Optimal parameters

On Delta, we experimented with finding the optimal parameters for the percentiles p_{tram} and p_{pq} using one node and our random graph generation. We tested values from 0.05 to 0.999 at an interval of 0.05 for both thresholds. For p_{tram}, we found that the optimal value is equal to 0.999, meaning that it is best to send all updates through tramlib instantly, rather than using the tram hold. We also found that the optimal value for p_{pq} is 0.05, meaning that very few updates should be added to pq instantly, while most should be placed into

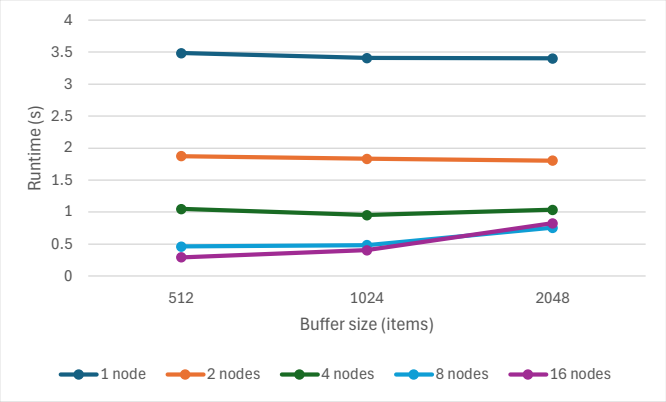

Fig. 6. The effect of the maximum number of items in tramlib sending buffers (triggering an automatic flush when exceeded) at various node counts: for more parallelism, smaller buffers are preferred.

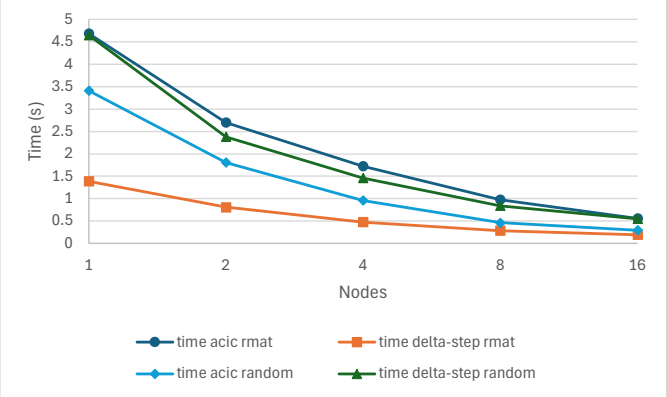

Fig. 7. The execution time of ACIC vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs, scale=26: ACIC is faster on random graphs, but slower on RMAT graphs.

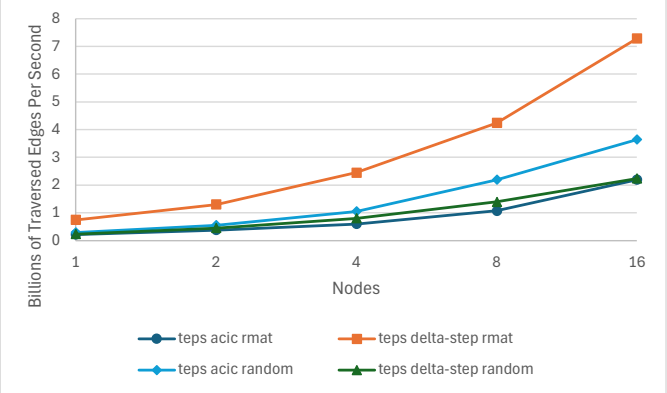

Fig. 8. The traversed edges per second (TEPS) of ACIC vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs: ACIC has higher TEPS on random graphs, but fewer on RMAT graphs.

pq_hold. The reason that a low p_{pq} percentile is effective, as shown in fig. 5, is that a low threshold effectively reduces the number of generated updates. The high optimal value of p_{tram} is because if an update is sub-optimal and improves a vertex distance, the pq is already effective in reducing the probability that updates will be generated from the sub-optimal update, and setting p_{tram} to a lower, tighter value only serves to slow down the messaging process by holding updates in $tram\_hold$ in the sending PE, instead of being quickly processed by the receiving PE.

We also experimented with the optimal size of sending buffers in tramlib. These experiments were run on 1 to 16 nodes on Frontier, to show how the optimal buffer size changes with available parallelism. Tramlib supports three buffer sizes: 512, 1024, and 2048 items. According to fig. 6, the optimal buffer size is 2048 for 1 and 2 nodes, 1024 for 4 and 8 nodes, and 512 for 16 nodes. This optimal value decreases as parallelism increases because more parallelism means tramlib maintains more sending buffers, since there are more PEs and processes that messages could be sent to. This means that provided that the size of the input graph is constant, more parallelism means each sending buffer takes longer to fill up on average, reducing the frequency of automatic buffer flushes when buffers are full, and therefore increasing the latency of a given message.

F. ACIC vs. Δ-stepping

We compare ACIC to a modified Δ-stepping algorithm on two types of graphs: one graph with edges randomly assigned between vertices, and one graph generated using the RMAT method to assign edges. We use the results from our tramlib buffer size tests to set the optimal buffer size for each node count for the experiments in this section. For the random graph, according to fig. 7, 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, ACIC generates 8-16% fewer updates than the Δ-stepping implementation, and 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, when using RMAT generation, 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. Like with the random graph, ACIC has 4-18% fewer updates than Δ-stepping.

ACIC is superior on random graphs, since when edges are assigned to origins and destinations and each vertex is equally likely to be assigned at either end of an edge, the distribution of final vertex distances from 0 to the maximum vertex distance is relatively even. This means that since ACIC minimizes wasteful work, the final distance of each vertex is set much sooner than in Δ-stepping, and the algorithm

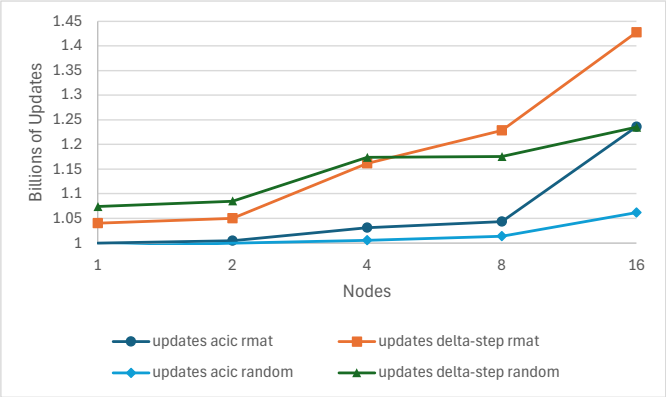

Fig. 9. The number of updates, or edge relaxations, of the Charm++ asynchronous SSSP algorithm vs. the RIKEN Δ-stepping algorithm for RMAT and random graphs: for both type of graphs, Charm++ generates fewer updates.

terminates sooner. However, RMAT graphs have a power-law degree distribution, where a small number of vertices have a high degree, while most vertices have very few neighbors. This means that RMAT graphs have a "tail" of vertices that are far away from the source vertex, and when an SSSP algorithm traverses these vertices, very little concurrency is available. ACIC does not process this tail efficiently compared to the Δ-stepping approach for a few reasons. First, the 1-dimensional partitioning used by ACIC creates a load imbalance that slows down graph processing compared to the 2D partitioning used by the RIKEN Δ-stepping algorithm. Additionally, ACIC is reliant on a cycle of reductions and broadcasts to advance thresholds and flush tramlib buffers, and during the tail portion of the algorithm, updates waste time in various application-level and tramlib buffers waiting for a broadcast to trigger a message advance. The RIKEN Δ-stepping algorithm uses a heuristic to change to using the Bellman-Ford algorithm once a local maxima of finalized vertices at a given Δ-stepping phase is reached. This optimization is designed to allow for fast processing of the tail portion of a graph, and our method of increasing all thresholds when we detect low parallelism is not as effective as the hybrid Δ-stepping algorithm.

V. Future Work

One of the current shortcomings of ACIC is the 1D graph partitioning that results in load imbalance across PEs and a lack of locality, resulting in potentially large message latencies. One alternative is the 2D partitioning used by the hybrid Δ-stepping algorithm [12], which divides the adjacency matrix of the input graph in two dimensions across the available processors, which helps improve locality. Communication only occurs within rows and within columns, which reduces the latency of messages. Another alternative is 1.5D partitioning [3], which classifies vertices into three categories: extremely high-degree, high-degree, and low-degree. The graph is then split into six distributed subgraphs based on the edges between one of these categories, ensuring high locality from all processors to vertices with the highest degrees.

Another way to deal with load imbalance is to fix the way updates are handled. At the application level, every update must be created by the sending PE containing the partition of the source vertex. However, any PE within the same process as the spending PE can calculate the PE destination of the update, and place it in a sending buffer or in $tram\_hold$. This type of work-stealing is helpful when load imbalance causes one PE to generate updates while others are idle, and Charm++ supports work-stealing queues shared by PEs on the same process.

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.

A further area of exploration would be refining the logic of our reductions and threshold decisions. Currently, when deciding on a percentile to calculate t_{tram} or t_{pq}, there are only two cases: one when the number of active updates is $100 * |PE|$ or less, and one when it is more. Instead of this two-tier logic, an ideal approach would be to create a function with a whole histogram as input and thresholds as output, taking into account both the number of updates and the shape of the histogram.

We would also like to test and tune ACIC for multiple types of graphs. Only RMAT and random graphs were tested in this paper, but there are other types of graphs that may particularly benefit from an asynchronous SSSP approach. One such class of graphs are 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. An asynchronous approach could show a substantial improvement, since asynchrony would allow such paths to be quickly traversed without any synchronization.

Finally, we would like to use the concepts developed in this paper to implement asynchronous algorithms for other graph problems. One candidate is the connected components problem for random graphs [7], where asynchronous reductions may be used to communicate information about vertices and components concurrently with computation.

VI. Conclusion

We present ACIC, a new asynchronous algorithm with various techniques to speed up the SSSP problem, including overlapping computation with a cycle of reductions and broadcasts to monitor the state of the algorithm and control the flow of updates. We analyze the effects of our techniques on various graph types, and show that we improve on a state-of-the-art synchronous algorithm on random graphs. Finally, we explore 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] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Edsger Wybe Dijkstra: his life, work, and legacy*, pages 287–290, 07 2022.

[7] P ERDdS and A R&wi. On random graphs i. *Publ. math. debrecen*, 6(290-297):18, 1959.

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

[9] 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.

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

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

[12] 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.

[13] 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.

[14] 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)

VII. 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

VIII. 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>
```