STREAMER: a Distributed Framework for Incremental Closeness Centrality Computation

Ahmet Erdem Sarıyüce\textsuperscript{1,2}, Erik Saule\textsuperscript{1}, Kamer Kaya\textsuperscript{1}, Ümit V. Çatalyurek\textsuperscript{1,3}

Depts. \textsuperscript{1}Biomedical Informatics, \textsuperscript{2}Computer Science and Engineering, \textsuperscript{3}Electrical and Computer Engineering

The Ohio State University

Email: sarıyuce.1@osu.edu, \{esaule,kamer,umit\}@bmi.osu.edu

Abstract—Networks are commonly used to model the traffic patterns, social interactions, or web pages. The nodes in a network do not possess the same characteristics: some nodes are naturally more connected and some nodes can be more important. Closeness centrality (CC) is a global metric that quantifies how important is a given node in the network. When the network is dynamic and keeps changing, the relative importance of the nodes also changes. The best known algorithm to compute the CC scores makes it impractical to recompute them from scratch after each modification. In this paper, we propose STREAMER, a distributed memory framework for incrementally maintaining the closeness centrality scores of a network upon changes. It leverages pipelined and replicated parallelism and takes NUMA effects into account. It speeds up the maintenance of the CC of a real graph with 916K vertices and 4.3M edges by a factor of 497 using a 64 nodes cluster.

I. INTRODUCTION

How central a node is in a network? Which nodes are more important during an entity dissemination? Centrality metrics have been used to answer such questions. They have been successfully used to carry analysis for various purposes such as power grid contingency analysis [10], quantifying importance in social networks [15], analysis of covert networks [12], decision/action networks [4], and even for finding the best store locations in cities [17]. As the networks became large, efficiency became a crucial concern while analyzing these networks. The algorithm with the best asymptotic complexity to compute the closeness and betweenness metrics [2] is believed to be asymptotically optimal [11]. And the research on fast centrality computation have focused on approximation algorithms [3], [6], [16] and high performance computing techniques [14], [19]. Today, the networks to be analyzed can be quite large, and we are always in a quest for faster techniques which help us to perform centrality-based analysis.

Many of today’s networks are dynamic. And for such networks, maintaining the exact centrality scores is a challenging problem which has been studied in the literature [7], [13], [18]. The problem can also arise for applications involving static networks such as the power grid contingency analysis and robustness evaluation of a network. The findings of such analyses and evaluations can be very useful to be prepared and take proactive measures if there is a natural risk or a possible adversarial attack that can yield undesirable changes on the network topology in the future. Similarly, in some applications, one might be interested in trying to find the minimal topology modifications on a network to set the centrality scores in a controlled manner. (Applications include speeding-up or containing the entity dissemination, and making the network immune to adversarial attacks).

Offline CC computation can be expensive for large-scale networks. Yet, one could hope that the incremental graph modifications can be handled in an inexpensive way. Unfortunately, as Fig. 1 shows, the effect of a local topology modification can be global. In a previous study, we proposed a sequential incremental closeness centrality algorithm which is orders of magnitude faster than the best offline algorithm [18]. Still, the algorithm was not fast enough to be used in practice. In this paper, we present STREAMER, a framework to efficiently parallelize the incremental CC computation on high-performance clusters.

STREAMER employs DataCutter [1], our in-house data-flow programming framework for distributed memory systems. In DataCutter, the computations are carried by independent computing elements, called filters, that have different responsibilities and operate on data passing through them. There are three main advantages of this scheme: first, it exposes an abstract representation of the application which is decoupled from its practical implementation. Second, the coarse-grain data-flow programming model allows replicated parallelism by instantiating a given filter multiple times so that the work can be distributed among the instances to improve the
parallelism of the application and the systems performance. And third, the execution is pipelined, allowing multiple filters to compute simultaneously on different iterations of the work. This pipelined parallelism is very useful to achieve overlapping of communication and computation. The best available algorithm for the offline centrality computation is pleasingly parallel (and scalable if enough memory is available) since it involves computation is pleasingly parallel (and scalable if enough ping of communication and computation.

And third, the execution is pipelined, allowing multiple filters parallelism of the application and the systems performance.

The previous results are ignored and the same distribute the SSSPs to the nodes and gather their results. Significant overlap among these computation/communication model and pipelined parallelism are very useful to achieve overlap-
tion problem which employs a pipelined parallelism

blocks in the online approach. And it is not trivial to obtain
an efficient parallelization of the incremental algorithm.

Our contributions can be summarized as follows:

1) We propose the first distributed-memory framework
STREAMER for the incremental centrality computation
problem which employs a pipelined parallelism
to achieve computation-computation and communication-overlap.

2) The worker nodes we used in the experiments have 8
cores. In addition to the distributed-memory parallelization,
we also leverage the shared-memory parallelization
and take NUMA effects into account.

3) The framework appears to scale linearly: when 63
worker nodes (8 cores/node) are used, for the networks
amazon0601 and web-Google, STREAMER obtains
456 and 497 speedups, respectively, compared to a single
worker node-single thread execution.

The paper is organized as follows: Section II introduces
the notation, formally defines the closeness centrality metric,
and describes the incremental approach in [18]. Section III
describes the proposed distributed framework for incremental
centrality computations in detail. The experimental analysis is
given in Section IV, and Section V concludes the paper.

II. INCREMENTAL CLOSENESS CENTRALITY

Let $G = (V, E)$ be a network modeled as a simple
undirected graph with $n = |V|$ vertices and $m = |E|$ edges
where each node is represented by a vertex in $V$, and a node-
ode interaction is represented by an edge in $E$. Let $\Gamma_G(v)$ be the set of vertices which are connected to $v$.

A graph $G' = (V', E')$ is a subgraph of $G$ if $V' \subseteq V$ and $E' \subseteq E$. A path is a sequence of vertices such that there exists an edge between consecutive vertices. Two vertices $u, v \in V$ are connected if there is a path from $u$ to $v$. If all vertex pairs are connected we say that $G$ is connected. If $G$ is not connected, then it is disconnected and each maximal connected subgraph of $G$ is a connected component, or a component, of $G$. We use $d_G(u, v)$ to denote the length of the shortest path between two vertices $u, v$ in a graph $G$. If $u = v$ then $d_G(u, v) = 0$. And if $u$ and $v$ are not connected $d_G(u, v) = \infty$.

Given a graph $G = (V, E)$, a vertex $v \in V$ is called an articulation vertex if the graph $G - v$ has more connected components than $G$. $G$ is biconnected if it is connected and it does not contain an articulation vertex. A maximal biconnected subgraph of $G$ is a biconnected component.

A. Closeness centrality

The farness of a vertex $u$ in a graph $G$ is defined as $\text{far}[u] = \sum_{v \in V} d_G(u, v)$. And the closeness centrality of $u$ is defined as $cc[u] = \frac{1}{\text{far}[u]}$. If $u$ cannot reach any vertex in the graph, then $cc[u] = 0$.

For a graph $G = (V, E)$ with $n$ vertices and $m$ edges, the complexity of the best $cc$ algorithm is $O(n(m + n))$ (Algorithm 1). For each vertex $s \in V$, it executes a Single-Source Shortest Paths (SSSP), i.e., initiates a breadth-first search (BFS) from $s$ and computes the distances to the connected vertices. And, as the last step, it computes $cc[s]$. Since a BFS takes $O(m + n)$ time, and $n$ SSSPs are required in total, the complexity follows.

<table>
<thead>
<tr>
<th>Algorithm 1: Offline centrality computation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Data:</strong> $G = (V, E)$</td>
</tr>
<tr>
<td><strong>Output:</strong> $cc[.]$</td>
</tr>
<tr>
<td>1 for each $s \in V$ do</td>
</tr>
<tr>
<td>$dSSSP(G, s)$ with centrality computation</td>
</tr>
<tr>
<td>$Q \leftarrow$ empty queue</td>
</tr>
<tr>
<td>$d[v] \leftarrow \infty, \forall v \in V \setminus {s}$</td>
</tr>
<tr>
<td>$Q.push(s), d[s] \leftarrow 0$</td>
</tr>
<tr>
<td>$\text{far}[s] \leftarrow 0$</td>
</tr>
<tr>
<td>while $Q$ is not empty do</td>
</tr>
<tr>
<td>$v \leftarrow Q.pop()$</td>
</tr>
<tr>
<td>for all $w \in \Gamma_G(v)$ do</td>
</tr>
<tr>
<td>if $d[w] = \infty$ then</td>
</tr>
<tr>
<td>$Q.push(w)$</td>
</tr>
<tr>
<td>$d[w] \leftarrow d[v] + 1$</td>
</tr>
<tr>
<td>$\text{far}[w] \leftarrow \text{far}[s] + d[w]$</td>
</tr>
<tr>
<td>$cc[s] = \frac{1}{\text{far}[s]}$</td>
</tr>
<tr>
<td>return $cc[.]$</td>
</tr>
</tbody>
</table>

B. Incremental closeness centrality

Algorithm 1 is an offline algorithm: it computes the CC scores from scratch. But today’s networks are dynamic and their topologies are changing through time. Centrality computation is an expensive task, and especially for large scale networks, an offline algorithm cannot cope with the
changing network topology. Hence, especially for large-scale, dynamic networks, online algorithms which do not perform the computation from scratch but only update the required scores in an incremental fashion are required. In a previous study, we used a set of techniques such as level-based work filtering and special-vertex utilization to reduce the centrality computation time for dynamic networks [18].

C. Level-based work filtering

The level-based filtering aims to reduce the number of SSSPs in Algorithm 1. Let \( G = (V, E) \) be the current graph and \( uv \) be an edge to be inserted. Let \( G' = (V, E \cup \{uv\}) \) be the updated graph. The centrality definition implies that for a vertex \( s \in V \), if \( d_G(s, t) = d_{G'}(s, t) \) for all \( t \in V \) then \( cc[s] = cc'[s] \). The following theorem is used to filter the SSSPs of such vertcices.

**Theorem 2.1 (Sarıyüce et al. [18]):** Let \( G = (V, E) \) be a graph and \( u \) and \( v \) be two vertices in \( V \) s.t. \( uv \notin E \). Let \( G' = (V, E \cup \{uv\}) \). Then \( cc[s] = cc'[s] \) if and only if \( |d_G(s, u) - d_G(s, v)| \leq 1 \).

Many interesting real-life networks are scale free. The diameters of a scale-free network is small, and when the graph is modified with minor updates, it tends to stay small. These networks also obey the power-law degree distribution. The level-based work filter is particularly efficient on these kind of networks. Figure 2 (top) shows the three cases while an edge \( uv \in E \) is being added to \( G \): \( d_G(s, u) = d_G(s, v) \), \( |d_G(s, u) - d_G(s, v)| = 1 \), and \( |d_G(s, u) - d_G(s, v)| > 1 \). Due to Theorem 2.1, an SSSP is required in Algorithm 1 only for the last case, since for the first two cases, the closeness centrality of \( s \) does not change. As Figure 2 (bottom) shows, the probability of the last case is less than 20% for three social networks used in the experiments. Hence, more than 80% of the SSSPs are avoided by using level-based filtering.

![Fig. 2. Three possible cases when inserting uv](image)

When an edge \( uv \) is inserted/deleted, to employ the filter, we first compute the distances from \( u \) and \( v \) to all other vertices. Detailed explanation can be found in [18].

D. Special-vertex utilization

The work filter can be assisted by employing and maintaining a biconnected component decomposition (BCD) of \( G \). A BCD is a partitioning \( \Pi(e) \) of the edge set \( E \) where \( \Pi(e) \) is the component of each edge \( e \in E \). A toy graph and its BCDs before and after an edge insertion are given in Fig. 3.

![Fig. 3. A graph G (left), its biconnected component decomposition II (middle), and the updated II’ after the edge bd is inserted (right).](image)

Let \( uv \) be the edge inserted to \( G = (V, E) \) and the final graph be \( G' = (V, E' \cup \{uv\}) \). Let \( \text{far} \) and \( \text{far}' \) be the farness scores of all the vertices in \( G \) and \( G' \). If the intersection \( \{\text{far}(uv) : w \in G(u)\} \cap \{\text{far}(uv) : w \in G(v)\} \) is not empty, there must be only one element in it (otherwise \( \Pi(e) \) is not a valid BCD), \( cid \), which is the id of the biconnected component of \( G' \) containing \( uv \). In this case, updating the BCD is simple: \( \Pi' = \Pi'(e) \) for all \( e \in E \) and \( \Pi'(uv) \) is set to \( cid \). If the intersection is empty (see the addition of \( bd \) in Fig. 3(b)), we construct \( \Pi' \) from scratch and set \( cid = \Pi'(uv) \) (e.g., \( cid = 2 \) in Fig. 3(c)). A BCD can be computed in linear, \( O(n + m) \) time [9]. Hence, the cost of BCD maintenance is negligible compared to the cost of updating closeness centrality.

Let \( G'_{cid} = (V_{cid}, E_{cid}') \) be the biconnected component of \( G' \) containing \( uv \). Let \( A_{cid} \subseteq V_{cid} \) be the set of articulation vertices of \( G' \) in \( G'_{cid} \). Given \( \Pi' \), it is easy to find the articulation vertices since \( u \in V \) is an articulation vertex if and only if it is at least in two components in the BCD: \(|\{\Pi'(uv) : uv \in E'\}| > 1 \).

The incremental algorithm executes SSSPs only for the vertices in \( G'_{cid} \). The contributions of the vertices in \( V \setminus V_{cid} \) are integrated to the SSSPs through their representatives \( rep : V \rightarrow V_{cid} \cup \{null\} \). For a vertex in \( V_{cid} \), the representative is itself. And for a vertex \( v \in V \setminus V_{cid} \), the representative is either an articulation vertex in \( A_{cid} \) or \( null \) if \( v \) and the vertices of \( V_{cid} \) are disconnected. Also, for all vertices \( x \in V \setminus V_{cid} \), we have \( \text{far}'[x] = \text{far}[x] + \text{far}[\text{rep}(x)] - \text{far}[\text{rep}(x)] \). Therefore, there is no need to execute SSSPs from these vertices. Detailed explanation and proofs are omitted for brevity and can be found in [18].

In addition to articulation vertices, we exploit the identical vertices which have the same/a similar neighborhood structure.
to further reduce the number of SSSPs. In a graph $G$, two vertices $u$ and $v$ are type-I-identical if and only if $\Gamma_G(u) = \Gamma_G(v)$. In addition, two vertices $u$ and $v$ are type-II-identical if and only if $\{u\} \cup \Gamma_G(u) = \{v\} \cup \Gamma_G(v)$. Let $u, v \in V$ be two identical vertices. One can easily see that for any vertex $w \in V \setminus \{u, v\}$, $d_G(u, w) = d_G(v, w)$. Therefore, if $\mathcal{I} \subseteq V$ is a set of (type-I or type-II) identical vertices, then the CC scores of all the vertices in $\mathcal{I}$ are equal.

We maintain the sets of identical vertices and while updating the CC scores of the vertices in $V$, we execute an SSSP for a representative vertex from each identical-vertex set. We then use the computed score as the CC score of the other vertices in the same set. The filtering is straightforward and the modifications on the algorithm are minor. When an edge $uv$ is added/removed to/from $G$, to maintain the identical vertex sets, we first subtract $u$ and $v$ from their sets and insert them to new ones. Candidates for being identical vertices are found using a hash function and the overall cost of maintaining the data structure is $O(n + m)$ [18].

III. STREAMER

STREAMER follows the component-based programming paradigm which has been used to describe and implement complex applications by way of components - distinct tasks with well-defined interfaces. By describing these components and the explicit data connections between them, the applications are decomposed along natural task boundaries according to the application domain. Therefore, the component-based application design is an intuitive process with explicit demarcation of task responsibilities. Furthermore, the communication patterns are also explicit; each component includes its input data requirements and outputs in its description.

STREAMER is written in DataCutter, our in-house component-based middleware tool which supports filter-stream programming, an instance of component-based programming. The filter-stream programming model [1] (a specific implementation of the dataflow programming model [5]) implements the computations as a set of components, referred as filters, that exchange data through logical streams. A stream denotes a uni-directional data flow from some filters (i.e., the producers) to others (i.e., the consumers). Data flows along these streams in untyped databuffers so as to minimize various system overheads. A layout is a filter ontology which describes the set of application tasks, streams, and the connections required for the computation.

Filter-stream programming enables some runtime benefits, which come at no additional cost to the developer. Applications composed of a number of individual tasks can be executed on parallel and distributed computing resources and gain extra performance over those run on strictly sequential machines. This is achieved by specifying a placement which is an instance of a layout with a mapping of the filters onto physical processors. A filter can be replicable, if it is stateless; for instance, if a filter’s output for a given databuffer does not depend on the ones it processed previously, it is stateless and replicable. A replicated filter can be placed on multiple processors to increase the throughput of the system.

Additionally, provided the interfaces exposed by a task to the rest of the application match, different implementations of tasks, possibly on different processor architectures can co-exist in the same application deployment, allowing developers to take full advantage of modern, heterogeneous supercomputers. Figure 4 shows an example filter-stream layout and placement. In this work, we used both distributed- and shared-memory architectures. However, thanks to filter-stream programming model, many-core systems such as GPUs and accelerators can also be used easily and efficiently if desired [8].

[Diagram of filter-stream layout and placement]

A. Pipelined parallelism

One of the DataCutter’s strengths is that it enables pipelined parallelism, where multiple stages of the pipeline (such as $A$ and $B$ in the layout in Fig. 4) can be executed simultaneously, and replicated parallelism can be used at the same time if some computation is stateless (such as filter $C$ in the same figure).

While computing the CC scores, the main portion of the computation comes from performing SSSPs for the vertices whose scores need to be updated. If there are many updates (we use the term “update” to refer to the SSSP operation which updates the CC score of a vertex), that part of the computation should occupy most of the machine. A typical synchronous decomposition of the application makes the work filtering of a Streaming Event (handling a single edge change) wait for the completion of all the work incurred by a previous Streaming Event. Since the worker nodes will wait for the work filtering to be completed, there can be a large waste of resources. We argue that the pipelined parallelism should be used to overlap the process of filtering the work and computing the updates on the graph.

We propose to use the four-filter layout shown in Fig. 5. The first filter is the $InstanceGenerator$ which first sends the initial graph to all the other filters. It then sends the Streaming Events as 4-tuples $(t, oper, u, v)$ to indicate that edge $uv$ has been either added or removed (specified by oper) at a given time $t$. (In the following, we only explain the system for edge insertion, but it is essentially the same for an edge removal.) In a real world application, this filter would be listening on the
network for topology modifications; but in our experiments, all the necessary information is read from a file.

StreamingMaster is responsible for the work filtering after each network modification. Upon inserting \( uv \) at time \( t \), it first computes the shortest distances from \( u \) and \( v \) to all other vertices at time \( t - 1 \). Then, it adds the edge \( uv \) into its local copy of the graph and updates the identical vertex sets as described in Section II-D. It partitions the edges of the graph to its biconnected components by using the algorithm in [9] and finds the component containing \( uv \). For each vertex \( s \in V \), it decides whether its CC score needs to be recomputed by checking the following conditions: (1) \( d(s, u) \) and \( d(s, v) \) differ by at least 2 units at time \( t - 1 \), (2) \( s \) is adjacent to an edge which is also in \( uv \)'s biconnected component, (3) \( s \) is the representative of its identical vertex set. StreamingMaster then informs the Aggregator about the number of updates it will receive for time \( t \). Finally, it sends the list of SSSP requests to the ComputeCC filter, i.e., the corresponding source vertex ids whose CC scores need to be updated.

ComputeCC performs the real work and computes the new CC scores after each graph modification. It waits for work from StreamingMaster, and when it receives a CC update request under the form of a 2-tuple \((t, s)\) (update time and source vertex id), ComputeCC advances its local graph representation to time \( t \) by using the appropriate updates from InstanceGenerator. If there is a change on the local graph, the biconnected component of \( uv \) is extracted, and a concise information of the graph structure and the set of articulation vertices are updated (as described in [18]). Finally, the exact CC score \( cc[s] \) at time \( t \) is computed and sent to the Aggregator as a 3-tuple \((t, s, cc[s])\). ComputeCC can be replicated to fill up the whole distributed memory machine without any problem: as long as a replica reads the update requests in the order of non-decreasing time units, it is able compute the correct CC scores.

The Aggregator filter gets the graph at a time \( t \) from InstanceGenerator. Then, it obtains the number of updates for that time from StreamingMaster. It computes the identical vertex sets as well as the BCD. It gets the updated CC scores from ComputeCC. Due to the pipelined parallelism used in the system and the replicated parallelism of ComputeCC, it is possible that updates from a later time can be received; STREAMER stores them in a backlog for future processing. When a \((t, s, cc[s])\) tuple is processed, the CC score of \( s \) is updated. If \( s \) is the representative of an identical vertex set, the CC scores of all the vertices in the same set are updated as well. If \( s \) is an articulation point, then the CC scores of the vertices which are represented by \( s \) (and are not in the biconnected component of \( uv \)) are updated as well, by using the difference in the CC score of \( s \) between time \( t \) and \( t - 1 \). Since Aggregator needs to know the CC scores at time \( t - 1 \) to compute the centrality scores at time \( t \), the system must be bootstrapped: the system computes explicitly all the centrality scores of the vertices for time \( t = 0 \).

B. Exploiting the shared memory architecture

The main portion of the execution time is spent by the ComputeCC filter. Therefore, it is important to replicate this filter as much as possible. Each replica of the filter will end up maintaining its own graph structure and computing its own BCD. Modern clusters are hierarchical and composed of distributed memory nodes where each node contains multiple processors featuring multiple cores that share the same memory space. For instance, the nodes used in our experiments are equipped with two processors, each having 4 cores.

It is a waste of computational power to recompute the data structure on each core. But it is also a waste of memory. Indeed, the cores of a processor typically share a common last level of cache and using the same memory space for all the cores in a processor might improve the cache utilization. We propose to split the ComputeCC filter in two separate filters which is transparent to the rest of the system thanks to DataCutter being component-based. The Preparator filter constructs the decomposed graph for each Streaming Event it is responsible for. The Executor filter performs the real work on the decomposed graph. In DataCutter, the filters running on the same physical node act run in separate pthreads within the same MPI process making sharing the memory as easy as communicating pointers. The release of the memory associated with the decomposed graph is handled by atomically decreasing a counter by the Executor.

The decoupling of the graph management and the CC score computation allows to either creating a single graph representation on each distributed memory node or having a copy of the graph on each NUMA domain of the architecture. This is shown in Fig. 6.

IV. EXPERIMENTS

STREAMER runs on the Owens cluster in the Department of Biomedical Informatics at The Ohio State University. For the experiments, we used all the 64 computational nodes, each
Table I

<table>
<thead>
<tr>
<th>Name</th>
<th># updates</th>
<th>time(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>web-NotreDame</td>
<td>399,420</td>
<td>8.16</td>
</tr>
<tr>
<td>amazon0601</td>
<td>1,548,288</td>
<td>140.19</td>
</tr>
<tr>
<td>web-Google</td>
<td>1,632,804</td>
<td>3,666.14</td>
</tr>
<tr>
<td>soc-pokec</td>
<td>4,924,759</td>
<td>226.20</td>
</tr>
</tbody>
</table>

A. Performance results

Figure 7 shows the performance and scalability of the system in different configurations. The performance is expressed in number of updates per second. The framework obtains up to 11,000 updates/sec on amazon0601 and web-Google, 49,000 updates/sec on web-NotreDame, and more than 750 updates/sec on the largest tested graph soc-pokec. It appears to scale linearly on the graphs amazon0601 and web-Google, soc-pokec. For the first two graphs, it reaches a speedup of 456 and 497, respectively, with 63 nodes and 8 threads/node compared to the single node-single thread configuration. (The incremental centrality computation on soc-pokec with a single node was too long to run the experiment, but the system is clearly scaling well on this graph.) The last graph, web-NotreDame, does not exhibit a linear scaling and obtains a speedup of only 316.

B. Execution-log analysis

Here we discuss the impact of pipelined parallelism and the sub-linear speedup achieved on web-NotreDame. In Figure 8, we present the execution logs for that graph obtained while using 3, 15, and 63 worker nodes. Each log plot shows three data series: the times at which StreamingMaster starts to process the Streaming Events, the total number of updates sent by StreamingMaster, and the number of updates processed by the Executors collectively. The three different logs show what happens when the ratio of update produced and update consumed per second changes.

The first execution-log plot with 3 worker nodes (Fig. 8(a)) shows the amount of the updates emitted and processed as two perfectly parallel almost straight lines. This indicates that the runtime of the application is dominated by processing the updates. As the figure shows, the times at which the master starts processing the Streaming Events are not evenly distributed. As mentioned before, StreamingMaster starts filtering for the next Streaming Event as soon as it sends all the updates for the current one. In other words, the amount of updates emitted for a given Streaming Event can be read from the execution log as the difference of the y-coordinates of two consecutive “update emitted” points (the first line). In the first plot, we can see that 6 out of 50 Streaming Events (the ticks at the end of each partial tick-lines) incurred significantly much more updates
The experiments we conducted showed three things. STREAMER can scale up and efficiently utilize our entire experimental cluster. By taking the hierarchical composition of the architecture into account (64 nodes, 2 processors per node, 4 cores per processor) and not considering it as a regular distributed machine (a 512 processor MPI cluster),
we obtained 10% additional improvement. Furthermore, the pipelined parallelism proved to be extremely necessary while using a large amount of nodes in a concurrent fashion.

V. CONCLUSION

Maintaining the correctness of a graph analysis is important in today’s dynamic networks. Computing the closeness centrality scores from scratch after each graph modification is prohibitive, and even sequential incremental algorithms are too expensive for networks of practical relevance. In this paper, we proposed STREAMER, a distributed memory framework which guarantees the correctness of the CC scores, exploits replicated and pipelined parallelism, and takes the hierarchical architecture of modern clusters into account. Using STREAMER on a 64 nodes, 8 cores/node cluster, we reached a speedup of 497.

STREAMER scales well. However, despite we exposed pipelined parallelism, the system eventually reaches a point where the SSSPs initiated from each source are no longer the bottleneck. In the future, we will remedy this problem by making the StreamingMaster and Aggregator faster. In particular, the StreamingMaster can use replicated parallelism: each Streaming Event can be filtered independently. We observed that the Aggregator cost is dominated by the biconnected component decomposition which we plan to parallelize.

ACKNOWLEDGMENTS

This work was supported in parts by the DOE grant DE-FC02-06ER2775 and by the NSF grants CNS-0643969, OCI-0904809, and OCI-0904802.

REFERENCES