STREAMER: a Distributed Framework for Incremental Closeness Centrality Computation

Ahmet Erdem Sarıyüce[1,2], Erik Saule[1], Kamer Kaya[1], Ümit V. Çatalyürek[1,3]

Depts. [1]Biomedical Informatics, [2]Computer Science and Engineering, [3]Electrical and Computer Engineering
The Ohio State University
Email: *sariyuce.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).

978-1-4799-0898-1/13/$31.00 © 2013 IEEE

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.

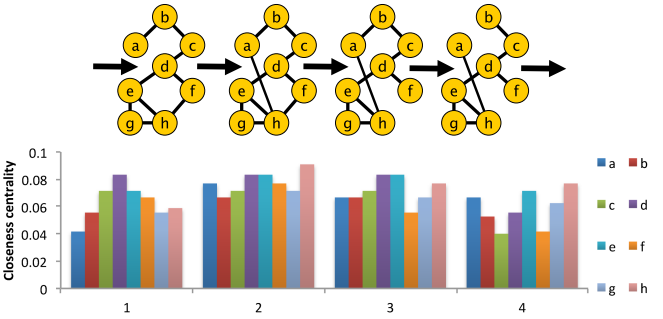


Fig. 1. A toy network with eight nodes, three consecutive edge (ah, fh, and ab, respectively) insertions/deletions, and CC scores.

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 n independent executions of the single-source shortest path (SSSP) algorithm [2]. In a naive distributed framework for the offline case, one can distribute the SSSPs to the nodes and gather their results. Here the computation is static, i.e., when the graph changes, the previous results are ignored and the same n SSSPs are re-executed. On the other hand, in the online approach, the updates can arrive at any time even while the centrality scores for a previous update are still being computed. Furthermore, the scores which need to be recomputed (the SSSPs that need to be executed) change w.r.t. the update. Finding these SSSPs and distributing them to the nodes is not a straightforward task. To be able to do that, the incremental algorithms maintain complex information such as the biconnected component decomposition of the current graph [18]. Hence, after each edge insertion/deletion, this information needs to be updated. There are several (synchronous and asynchronous) blocks in the online approach. And it is not trivial to obtain an efficient parallelization of the incremental algorithm. As our experiments will show, the data-flow programming model and pipelined parallelism are very useful to achieve a significant overlap among these computation/communication blocks and yield a scalable solution for the incremental centrality computation.

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 computation-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-node 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 $\texttt{far}[u] = \sum_{\substack{v \in V \\ d_G(u,v) \neq \infty}} d_G(u, v)$. And the closeness centrality of u is defined as $\texttt{cc}[u] = \frac{1}{\texttt{far}[u]}$. If u cannot reach any vertex in the graph, then $\texttt{cc}[u] = 0$.

For a graph $G = (V, E)$ with n vertices and m edges, the complexity of the best `cc` algorithm is $\mathcal{O}(n(m + n))$ (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 $\texttt{cc}[s]$. Since a BFS takes $\mathcal{O}(m+n)$ time, and n SSSPs are required in total, the complexity follows.

Algorithm 1: Offline centrality computation

```
Data: G = (V, E)
Output: cc[.]
for each s ∈ V do
    ▷SSSP(G, s) with centrality computation
    Q ← empty queue
    d[v] ← ∞, ∀v ∈ V \ {s}
    Q.push(s), d[s] ← 0
    far[s] ← 0
    while Q is not empty do
        v ← Q.pop()
        for all w ∈ Γ_G(v) do
            if d[w] = ∞ then
                Q.push(w)
                d[w] ← d[v] + 1
                far[s] ← far[s] + d[w]
    cc[s] = 1/far[s]
return cc[.]
```

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 $\texttt{cc}[s] = \texttt{cc}'[s]$. The following theorem is used to filter the SSSPs of such vertices.

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 $\texttt{cc}[s] = \texttt{cc}'[s]$ 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.

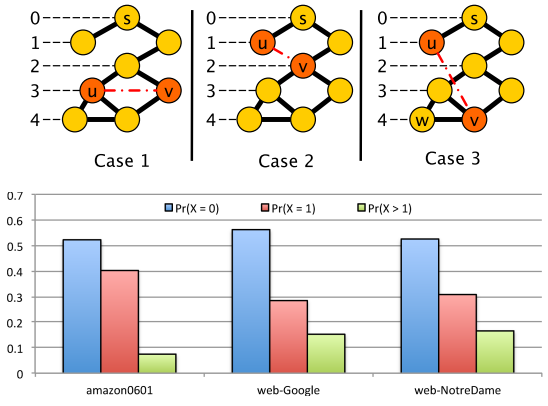
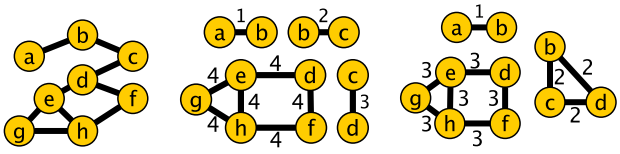


Fig. 2. Three possible cases when inserting uv: for each vertex s, one of the following is true: (1) $d_G(s,u) = d_G(s,v)$, (2) $|d_G(s,u) - d_G(s,v)| = 1$, or (3) $|d_G(s,u) - d_G(s,v)| > 1$ (top). The bars show the distribution of random variable $X = |d_G(u,w) - d_G(u,w)|$ into three cases while an edge uv is being added to G (bottom). For each network, the probabilities are computed by using $1,000$ random edges from E. For each edge uv, we constructed the graph $G = (V, E \setminus \{uv\})$ by removing uv from the final graph and computed $|d_G(s,u) - d_G(s,v)|$ for all $s \in V$.

Although Theorem 2.1 yields to a filter only in case of edge insertions, the same idea can easily be used for edge deletions.



(a) G (b) Π (c) Π'

Fig. 3. A graph G (left), its biconnected component decomposition Π (middle), and the updated Π' after the edge bd is inserted (right). The articulation vertices before and after the edge insertion are $\{b, c, d\}$ and $\{b, d\}$, respectively. After the addition, the second component contains the new edge, i.e., $cid = 2$. This component is extracted first, and the algorithm performs updates only for its vertices $\{b, c, d\}$. It also initiates a fixing phase to make the CC scores correct for the rest of the vertices.

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 Π 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.

Let uv be the edge inserted to $G = (V, E)$ and the final graph be $G' = (V, E' = E \cup \{uv\})$. Let $\texttt{far}$ and $\texttt{far}'$ be the farness scores of all the vertices in G and G'. If the intersection $\{\Pi(uw) : w \in \Gamma_G(u)\} \cap \{\Pi(vw) : w \in \Gamma_G(v)\}$ is not empty, there must be only one element in it (otherwise Π 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: $\Pi'(e)$ is set to $\Pi(e)$ 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 Π' from scratch and set $cid = \Pi'(uv)$ (e.g., $cid = 2$ in Fig. 3(c)). A BCD can be computed in linear, $\mathcal{O}(m + n)$ 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 $\mathcal{A}_{cid} \subseteq V_{cid}$ be the set of articulation vertices of G' in G'_{cid}. Given Π', 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: $|\{\Pi'(uw) : uw \in E'\}| > 1$.

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* $rep : V \rightarrow V_{cid} \cup \{\texttt{null}\}$. 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 $\mathcal{A}_{cid}$ or $\texttt{null}$ if v and the vertices ofV_{cid} are disconnected. Also, for all vertices $x \in V \setminus V_{cid}$, we have $\texttt{far}'[x] = \texttt{far}[x] + \texttt{far}'[rep(x)] - \texttt{far}[rep(x)]$. 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 $\mathcal{O}(n + m)$ [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].

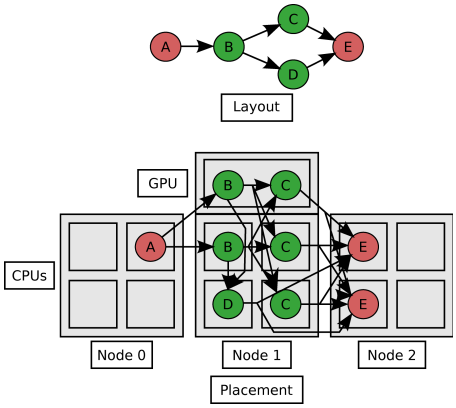


Fig. 4. A toy filter-stream application layout and its placement.

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

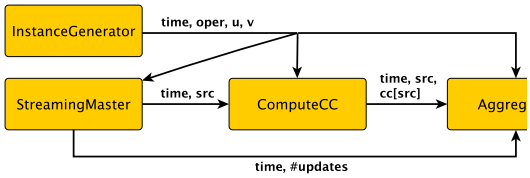


Fig. 5. Layout of STREAMER.

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 $\texttt{cc}[s]$ at time t is computed and sent to the *Aggregator* as a 3-tuple $(t,s,\texttt{cc}[s])$. *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 $(t,s,\texttt{cc}[s])$ 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

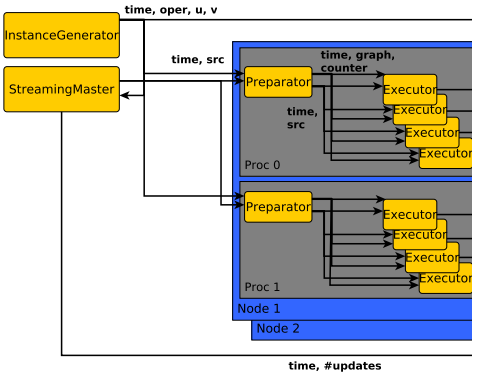


Fig. 6. Placement of STREAMER using 2 worker nodes with 2 quad-core processors. (The node 2 is hidden). The remaining filters are on node 0.

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

Name	$\|V\|$	$\|E\|$	# updates	time(s)
web-NotreDame	325,729	1,090,008	399,420	8.16
amazon0601	403,394	2,443,308	1,548,288	140.19
web-Google	916,428	4,321,958	2,527,088	226.20
soc-pokec	1,632,804	30,622,464	4,924,759	6,366.14

with dual Intel Xeon E5520 Quad-core CPUs (with 2-way Simultaneous Multithreading, and 8MB of L3 cache per processor), 48 GB of main memory. The nodes are interconnected with 20 Gbps InfiniBand. The algorithms were run on CentOS 6, and compiled with GCC 4.5.2 using the -O3 optimization flag. DataCutter uses an InfiniBand-aware MPI to leverage the high performance interconnect: here we used MVAPICH 1.1.

For testing purposes, we picked 4 large social network graphs from the SNAP dataset to perform the test at scale. The properties of the graphs are summarized in Table **??**. For simulating the addition of the edges, we removed 50 edges from the graphs and added them back one by one. The streamed edges were selected randomly and uniformly. For comparability purposes, all the runs performed on the same graph use the same set of edges. The number of updates induced by that set of edges when applying filtering using identical vertices, biconnected component decomposition, and level filtering is given in Table **??**. In the experiments, the data comes from a file, and the Streaming Events are pushed to the system as quickly as possible so as to stress the system.

All the results presented in this section are extracted from a single run of STREAMER with proper parameters. The regularity in the plots indicates there would be a small variance on the runtimes, which induces a reasonable confidence in the significance of the quoted numbers. In the experiments, *StreamingMaster* and *Aggregator* run on the same node, apart from all the computational filters. Therefore, we report the number of worker nodes, but an extra node is always used.

To give an idea of the actual amount of computation, in the last column of Table **??**, we report the time STREAMER spends to update the CC scores upon 50 edge insertions by using all 63 worker nodes. We present the parallel time and not the sequential time for two reasons: (1) Our framework is never really sequential, even using a single *ComputeCC* filter would not actually be sequential. (2) The sequential runtime on the biggest tested graph (soc-pokec) is prohibitive (estimated at about a month). As all the execution times given in this section, the times in Table **??** do not contain the initialization time. That is the time measurement starts once STREAMER is idle, waiting to receive Streaming Events.

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.

TABLE I
THE PERFORMANCE OF STREAMER WITH 31 WORKER NODES AND DIFFERENT NODE-LEVEL CONFIGURATIONS NORMALIZED TO 1 THREAD CASE (PERFORMANCE ON soc-pokec IS NORMALIZED TO 8 THREADS, 1 GRAPH/THREAD). THE LAST COLUMN IS THE ADVANTAGE OF SHARED MEMORY AWARENESS (RATIO OF COLUMNS 5 AND 3).

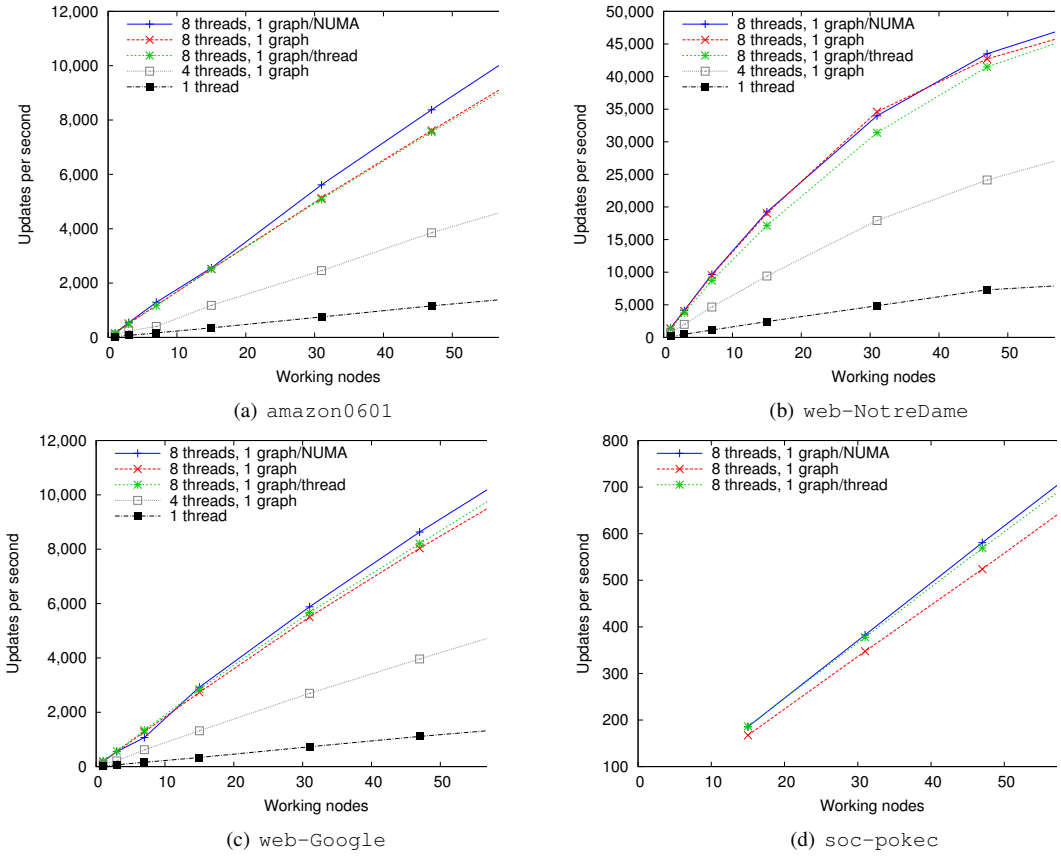
Name	4 threads	8 threads, 1 graph per			Shared Mem.
		thread	node	NUMA	awareness
web-NotreDame	3.69	6.46	7.13	6.99	1.08
amazon0601	3.26	6.75	6.81	7.45	1.10
web-Google	3.69	7.77	7.55	8.06	1.03
soc-pokec	-	1.00	0.92	1.01	1.01

Let us first evaluate the performance obtained under different node-level configurations. Table I presents the relative performance of the system using 31 worker nodes while using 1, 4, or 8 threads per node. When compared with the single thread configuration, using 4 threads (the second column) is more than 3 times faster, while using 8 threads (columns 3–5) per node usually gives 6.5 speedup or more. Overall, having multiple cores is fairly well exploited. Properly taking the shared-memory aspect of the architecture into account (column 5) brings a performance improvement between 1% to 10% (the last column). In one instance (web-Google with a graph for each NUMA domain), we observed that the normalized performance is more than the number of cores. This can be explained by the difference in the amount of work due to the distribution of the updates from different Streaming Events to the threads.

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



(a) amazon0601 (b) web-NotreDame

(c) web-Google (d) soc-pokec

Fig. 7. Scalability: the performance is expressed in the number of updates per second. Different worker-node configurations are shown. "8 threads, 1 graph/thread" means that 8 *ComputeCC* filters are used per node. "8 threads, 1 graph" means that 1 *Preparator* and 8 *Executor* filters are used per node. "8 threads, 1 graph/NUMA" means that 2 *Preparators* per node (one per NUMA domain) and 8 *Executors* are used.

than the others. While these events are being processed, the two lines stay straight and parallel, because in DataCutter, writing to a downstream filter is a buffered operation. Once the buffer is full, the operation becomes blocking.

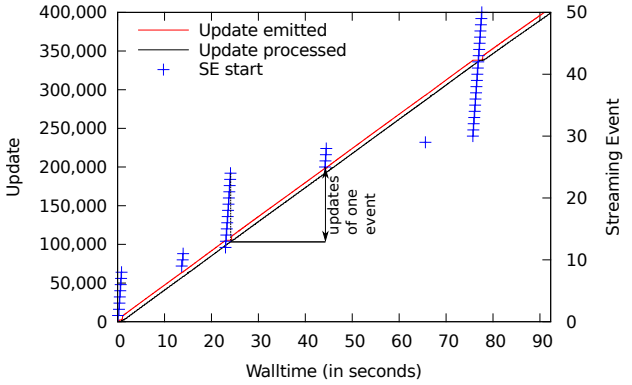
The second execution log with 15 worker nodes (Fig. 8(b)) shows a different picture. Here, the log is about 4 times shorter and the lines are not perfectly parallel. The number of updates emitted shows three plateaus for more than a second around times 0, 5, and 16 seconds. These plateaus exist because many consecutive Streaming Events do not generate a significant amount of updates; therefore, the master spends all its time by filtering the work for these Streaming Events.

The second plateau around time 5 seconds of the execution log with 15 worker nodes lasts 1.2 secs, and less than 100 updates are sent during that interval. However, as the plot shows, the worker nodes do not run out of work and process more than $25,000$ updates during the plateau. This is possible because the computation in STREAMER is pipelined. If the system were synchronous the worker nodes would spend most of that plateau waiting which yields a longer execution time and worse performance. In addition to the three large plateaus, cases with a few consecutive Streaming Events that lead to barely no updates are slightly visible around times 3 and 9. These two smaller cases are hidden by the pipelined parallelism. The third plateau is much longer than the second one (20 Streaming Events, 2.1 secs) and the worker nodes eventually run out of work halfway through the plateau. As can be seen in Fig. 7(b), the performance does not show linear scaling at 15 worker nodes; But it is still good, thanks to the pipelined parallelism.

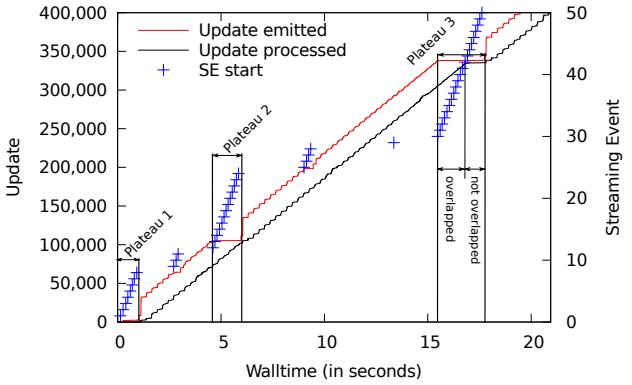
When 63 worker nodes are used, the execution log (Fig. 8(c)) presents another picture. With the increase on the workers' processing power, *StreamingMaster* is now the main bottleneck of the computation. Two additional, considerably large plateaus appeared, and *StreamingMaster* starts to spend more than half of its time with the work filtering. However, during these times, the workers keep processing the updates, but at varied rates, due to temporary work starvation. The work filtering and the actual work are being processed mostly simultaneously showing that pipelined parallelism is very effective in this situation. Without the pipelined parallelism, the computation time would certainly be 2 secs longer, and 25% worse performance would be achieved.

C. Summary of the experimental results

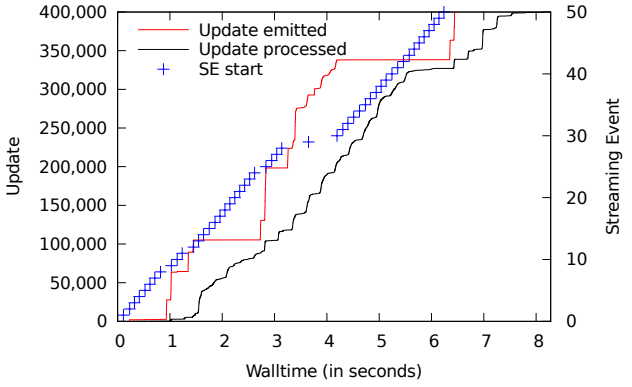
The experiments we conducted showed three things. STREAMER can scale up and efficiently utilize our entire experimental cluster. By taking the hierarchical composition of the architecture into account (64 nodes, 2 processors per node, 4 cores per processor) and not considering it as a regular distributed machine (a 512 processor MPI cluster),



(a) 3 worker nodes



(b) 15 worker nodes



(c) 63 worker nodes

Fig. 8. Execution logs for `web-NotreDame` on different number of nodes. Each plot shows the total number of updates sent by *StreamingMaster* and processed by the *Executors*, respectively (the two lines), and the times at which *StreamingMaster* starts to process Streaming Events (the set of ticks).

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 biconnnected 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

[1] M. D. Beynon, T. Kurç, Ü. V. Çatalyürek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[2] U. Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, 25(2):163–177, 2001.

[3] S. Y. Chan, I. X. Y. Leung, and P. Liò. Fast centrality approximation in modular networks. In *Proc. of CIKM-CNIKM*, 2009.

[4] Ö. Şimşek and A. G. Barto. Skill characterization based on betweenness. In *Proc. of NIPS*, 2008.

[5] J. B. Dennis. Data flow supercomputers. *Computer*, 13(11):48–56, 1980.

[6] D. Eppstein and J. Wang. Fast approximation of centrality. In *Proc. of SODA*, 2001.

[7] O. Green, R. McColl, and D. A. Bader. A fast algorithm for streaming betweenness centrality. In *Proc. of SocialCom*, 2012.

[8] T. D. R. Hartley, E. Saule, and U. V. Catalyurek. Improving performance of adaptive component-based dataflow middleware. *Parallel Computing*, 38(6-7):289–309, 2012.

[9] J. Hopcroft and R. Tarjan. Algorithm 447: efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, June 1973.

[10] S. Jin, Z. Huang, Y. Chen, D. G. Chavarría-Miranda, J. Feo, and P. C. Wong. A novel application of parallel betweenness centrality to power grid contingency analysis. In *Proc. of IPDPS*, 2010.

[11] S. Kintali. Betweenness centrality : Algorithms and lower bounds. *CoRR*, abs/0809.1906, 2008.

[12] V. Krebs. Mapping networks of terrorist cells. *Connections*, 24, 2002.

[13] M.-J. Lee, J. Lee, J. Y. Park, R. H. Choi, and C.-W. Chung. QUBE: a Quick algorithm for Updating BEtweenness centrality. In *Proc. of WWW*, 2012.

[14] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. G. Chavarría-Miranda. A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. In *Proc. of IPDPS*, 2009.

[15] E. L. Merrer and G. Trédan. Centralities: Capturing the fuzzy notion of importance in social graphs. In *Proc. of SNS*, 2009.

[16] K. Okamoto, W. Chen, and X.-Y. Li. Ranking of closeness centrality for large-scale social networks. In *Proc. of FAW*, 2008.

[17] S. Porta, V. Latora, F. Wang, E. Strano, A. Cardillo, S. Scellato, V. Iacoviello, and R. Messora. Street centrality and densities of retail and services in Bologna, Italy. *Environment and Planning B: Planning and Design*, 36(3):450–465, 2009.

[18] A. E. Sarıyüce, K. Kaya, E. Saule, and Ümit V. Çatalyürek. Incremental algorithms for network management and analysis based on closeness centrality. *CoRR*, abs/1303.0422, 2013.

[19] Z. Shi and B. Zhang. Fast network centrality analysis using GPUs. *BMC Bioinformatics*, 12:149, 2011.