Incremental Closeness Centrality in Distributed Memory

Ahmet Erdem Sarıyüce[1,2], Erik Saule[4], 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
[4] Dept. Computer Science - University of North Carolina at Charlotte
Email:*sariyuce.1@osu.edu, esaule@uncc.edu, kaya.20@osu.edu, 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, replicated parallelism and SpMM-based BFSs, and takes NUMA effects into account. It makes the CC maintenance of real-life networks with millions of interactions significantly faster and obtains almost linear speedups on a 64 nodes 8 threads/node cluster.

Keywords: Closeness centrality, Incremental centrality, BFS, Parallel programming, Cluster Computing

1. Introduction

How central is a node 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 [15], quantifying importance in social networks [21], analysis of covert networks [17], decision/action networks [8], and even for finding the best store locations in cities [24]. As the networks become large, efficiency becomes a crucial concern while analyzing these networks. The algorithm with the best asymptotic complexity to compute the closeness and betweenness metrics [4] is believed to be asymptotically optimal [16]. And the research on fast centrality computation have focused on approximation algorithms [7, 10, 22] and high performance computing techniques [20, 28]. 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 [11, 18, 25]. 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; for instance 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 Closeness Centrality (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 Figure 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 [25]. 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* [3], our in-house data-flow programming framework for

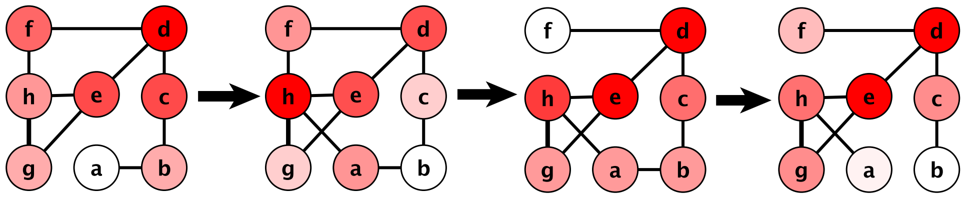


Figure 1: A toy network with eight nodes and three consecutive edge (ah, fh, and ab, respectively) insertions/deletions. The nodes are colored with respect to their relative CC scores where red implies a higher closeness score.

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 system's 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 [4]. 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 [25]. 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.

In this paper, we extend STREAMER that we introduced in [?]. Our contributions can be summarized as follows:

1. We propose the first distributed-memory framework STREAMER for the incremental closeness 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 scales linearly: when 63 worker nodes (8 cores/node) are used, for the networks `amazon0601`, `web-Google`, and `soc-pokec` STREAMER obtains almost linear speedups compared to a single worker node-single thread execution [?].
4. The framework is modular which makes it easily extendable. When the number of used nodes increases, the computation inevitably reaches a bottleneck on the extremities of the analysis pipeline which are not parallel. In [?], this effect appeared on one of the graph (`web-NotreDame`). We show how to the management of the computation can be made parallel by leveraging the modularity of dataflow middleware.
5. Using an SpMM-based BFS formulation, we significantly improved the incremental CC computation performance and show that the data-flow programming model makes STREAMER highly modular and easy to enhance with novel algorithmic techniques.
6. These new techniques provide an improvement of a factor between 2.2 to 9.3 times compared to the techniques presented in [?].

The paper is organized as follows: Section 2 introduces the notation, formally defines the closeness centrality metric, and describes the incremental approach in [25]. Section 3 describes the proposed distributed framework for incremental centrality computations in detail. The experimental analysis is given in Section 4, and Section 5 concludes the paper.

2. 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*.

2.1. Closeness centrality

The *farness* of a vertex $u \in V$ in a graph $G = (V, E)$ 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 $\texttt{cc}$ 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 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: $cc[.]$
1 for each $s \in V$ do
      ▷SSSP($G$, $s$) with centrality computation
      $Q \leftarrow$ empty queue
      $d[v] \leftarrow \infty, \forall v \in V \setminus \{s\}$
      $Q$.push($s$), $d[s] \leftarrow 0$
      far[$s$] $\leftarrow 0$
      while $Q$ is not empty do
          $v \leftarrow Q$.pop()
          for all $w \in \Gamma_G(v)$ do
              if $d[w] = \infty$ then
                  $Q$.push($w$)
                  $d[w] \leftarrow d[v] + 1$
                  far[$s$] $\leftarrow$ far[$s$] $+ d[w]$
      cc[$s$] = $\frac{1}{\text{far}[s]}$
  return cc[.]
```

2.2. *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 [25].

2.3. *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. [25]). *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.

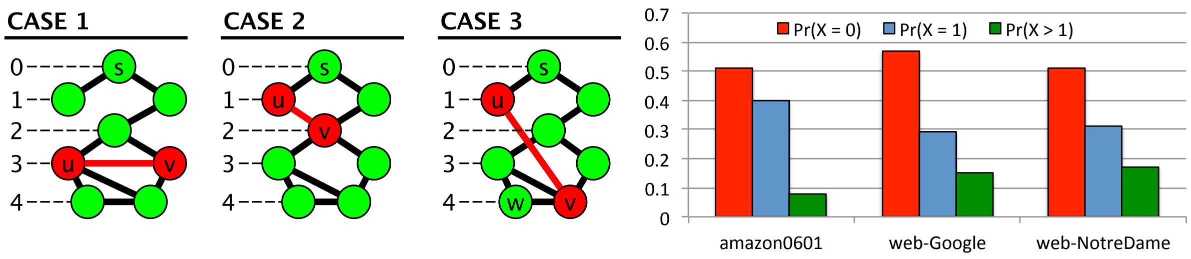
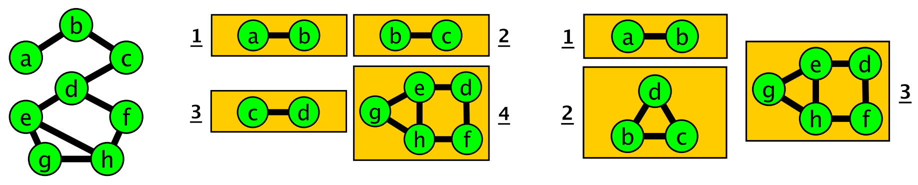


Figure 2: Three possible cases when inserting uv: for each vertex s, one of the following is true: (1) $d_G(s,u) = d_G(s,v)$, (2) $|d_G(s,u) - d_G(s,v)| = 1$, or (3) $|d_G(s,u) - d_G(s,v)| > 1$ (top). The bars show the distribution of random variable $X = |d_G(u,w) - d_G(u,w)|$ into three cases while 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. 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 [25].



(a) G (b) Π (c) Π'

Figure 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.

2.4. 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 [13]. 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 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 [25].

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 $\mathcal{O}(n+m)$ [25].

2.5. Simultaneous source traversal

The performance of sparse kernels is mostly hindered by irregular memory access. The most famous example for sparse computation is the multiplication of a sparse matrix by a dense vector (SpMV). Several techniques, like register blocking [6, 29] and usage of different matrix storage formats [2, 19], are proposed to regularize the memory access pattern. However, multiplying a sparse matrix by multiple vectors is the most efficient and popular technique to regularize the memory access pattern. Once the multiple vectors are organized as a dense matrix, the problem becomes the multiplication of a sparse matrix by a dense matrix (SpMM). Each nonzero of the sparse matrix causes the multiplication of a single element of the vector in SpMV, and it results in the multiplications of as many consecutive elements of the dense matrix as its number of columns in SpMM.

Accommodating that idea for closeness centrality computation turns out to be concurrently computing the multiple sources at the same time. However, as opposed to SpMV, in which the vector is dense and therefore each non-zero induces exactly one multiplication, in BFS, not all the non-zeros will induce operations. That is to say, a vertex in BFS may or may not be traversed depending on which level is currently being processed. Thus, the traditional queue-based implementation of BFS does not seem to be easily extendable to support multiple BFSs in a vector-friendly manner. We developed that idea in [26] and present here the main idea.

2.5.1. An SpMV-based formulation of closeness centrality

The idea is to convert to a more simple definition of level synchronous BFS: If one of the neighbor of v is part of level $\ell - 1$ and v is not part of any level $\ell' < \ell$, then vertex v is part of level ℓ. This formulation is used in parallel implementations of BFS on GPU [14, 23, 28], on shared memory systems [1] and distributed memory systems [5].

The algorithm is better represented using binary variables. Let x_i^ℓ be the binary variable that is `true` if vertex i is part of the frontier at level ℓ for a BFS. The neighbors of level ℓ is represented by a vector $y^{\ell+1}$ computed by $y_k^{\ell+1} = \mathtt{OR}_{j \in \Gamma(k)} x_j^\ell$. The next level is then computed with $x_i^{\ell+1} = y_i^{\ell+1}$ `AND not` $(\mathtt{OR}_{\ell' \leq \ell} x_i^{\ell'})$. Using these variables, one can increase the farness of the source by ℓ if i is at level ℓ (i.e., if $x^\ell = 1$). One can remark that $y^{\ell+1}$ is the result of the "multiplication" of the adjacency matrix of the graph by x^ℓ in the $(\mathtt{OR},\mathtt{AND})$ semi-ring.

2.5.2. An SpMM-based formulation of closeness centrality

It is easy to derive an algorithm from the formulation given above for closeness centrality computation that processes multiple sources at once. Instead of manipulating a single vector x and y where each element is a single bit, one can encode 32 vectors for 32 BFSs so that one *int* can encode the state of a single vertex across the 32 BFSs. The algorithm becomes quite efficient as it does not use more memory and process 32 BFS at once. All the operations become simple bit-wise `and`, `or` and `not`.

Theoretically, the asymptotic complexity changes when BFS is implemented using an SpMM approach. The complexity of the traditional queue-based BFS algorithm is $\mathcal{O}(|E|)$. If the adjacency matrix is stored row-wise, the SpMM-based implementation boils down to a bottom-up implementation of BFS which has a natural write access pattern. However, it becomes impossible to only traverse the relevant nonzero of the matrix and the complexity of the algorithm becomes $\mathcal{O}(|E| \times L)$, where L is the diameter of the graph. Social networks have small world properties which implies that their diameter is low and we do not feel that this asymptotic factor of L will hinder performance.

Moreover, multiple BFSs are performed simultaneously (here 32) which can recoup for the loss. In [26], the algorithm computes the impact of the sources on all the vertices of the graph. What we presented in this section does the reverse and compute the impact of all the vertices of the graph on the sources. Despite worse asymptotic complexity the performance of the SpMM approach outperforms traditional BFS approach [26]. Moreover, such algorithm is compatible with the decomposition of the graph in biconnected components [27] which can lead to further improvement. Because this algorithm computes the farness of the sources, it can be used to compute centrality incrementally.

3. 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 which supports filter-stream programming, an instance of component-based programming. The filter-stream programming model [3] (a specific implementation of the dataflow programming model [9]) 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, 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 [12].

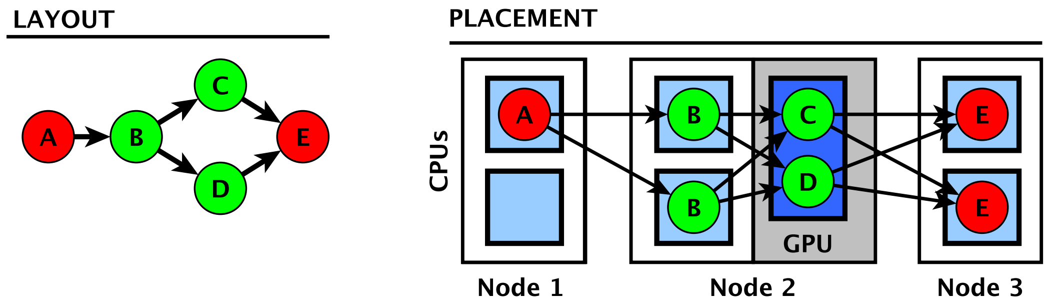


Figure 4: A toy filter-stream application layout and its placement.

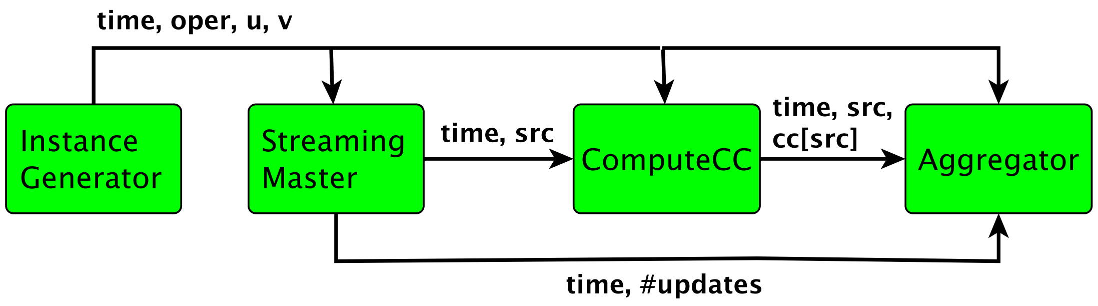


Figure 5: Layout of Streamer.

3.1. 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 B 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 2.4. It partitions the edges of the graph to its biconnected components by using the algorithm in [13] 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 [25]). 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 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 $t = 0$.

3.2. 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.

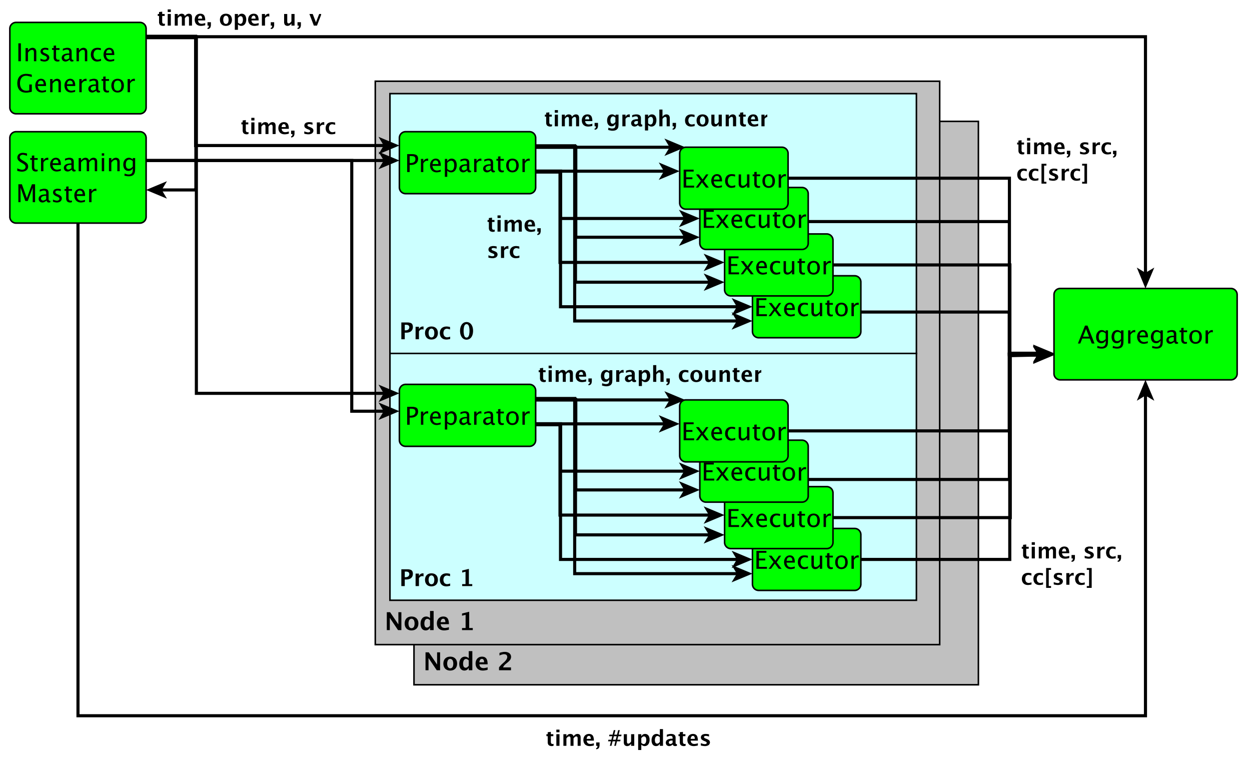


Figure 6: Placement of STREAMER using 2 worker nodes with 2 quad-core processors. (The node 2 is hidden). The remaining filters are on node 0.

3.3. Parallelizing StreamingMaster

When the number of cores used for *ComputeCC* increases the relative importance of *ComputeCC* in the total runtime decreases. Theoretically with an infinite number of cores for *ComputeCC*, the time required by it will drop to zero. And the bottleneck of the application becomes the rate at which *StreamingMaster* can generate updates request and the rate at which *Aggregator* can merge the computed results. To improve these rates, we follow the avenue replacing them with a construct that allow parallel execution.

StreamingMaster is decomposed in three filters which are laid out according to Figure 7. Most of the work done by *StreamingMaster* is done by a filter called *StreamingMaster* which supports replication. Each of the replica receives the list of edges it has to compute the filtering from a *WorkDistributor*. This *WorkDistributor* just listens on the modification on the graph and distribute the Streaming Events among the different *StreamingMaster*.

It is important that the *ComputeCC* receives the update requests in non-decreasing order

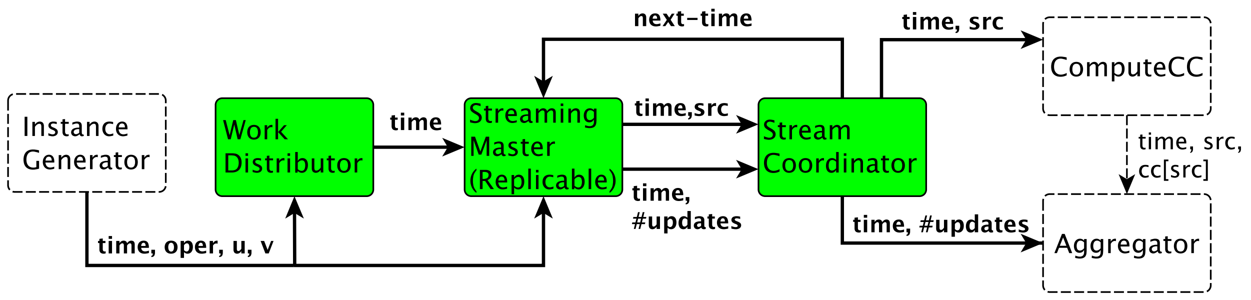


Figure 7: Replicating StreamingMaster for a better scaling when the number of processors is large.

of streaming events. *StreamCoordinator* is responsible for enforcing that order. *StreamCoordinator* sits between the *StreamingMaster* and the *ComputeCC* (and the *Aggregator*) and relays messages to them. The *StreamCoordinator* tells *StreamingMaster* which streaming event is the next one. In other words, before outputting the list of updates (and metadata for the *Aggregator*), the *StreamingMaster* reads from the *StreamCoordinator* whether it is time to output.

3.4. Parallelizing Aggregator

One of the challenges in parallelizing the *Aggregator* is that there can be only one filter that actually stores the centrality values of the network. Fortunately, most of the computation time spent by the *Aggregator* is spent in preparing the network rather than in applying the updates. We modify the layout of the *Aggregator* to match that of Figure 8.

Therefore only a single filter, we will call *Aggregator* for the sake of simplicity, is responsible for applying the updates, and is only responsible for this. It takes three kinds of input: the updates on the graph itself, the information of how many updates will be applied for each streaming event and information on the graph (the graph itself, its biconnected decomposition and identical vertices).

The information about the graph are constructed by another filter called *AggregatorPreparator* which can be replicated. It listens to the Streaming Events and receive work assignment. It then computes the sets of identical vertices and its biconnected component decomposition and send its downstream.

The work in the *AggregatorPreparator* is distributed in a way similar to the parallelization

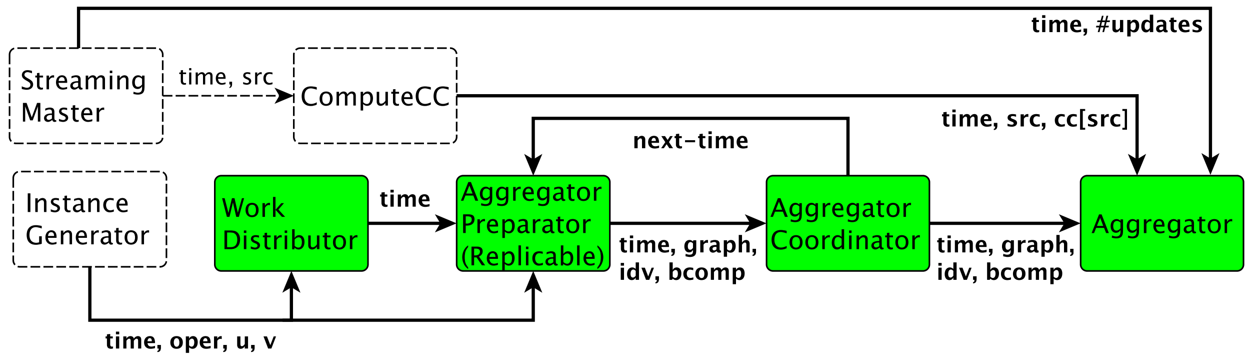


Figure 8: Replicating Aggregator for a better scaling when the number of processors is large.

of the *StreamingMaster*. Also the graph information must reach the *Aggregator* in the order of the Streaming Event. An *AggregatorCoordinator* is used to regulate the order in which the graph information are sent. It behaves under the same principle as *StreamCoordinator*.

4. 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 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 4.8.1 using the -O3 optimization flag. DataCutter uses an InfiniBand-aware MPI to leverage the high performance interconnect: here we used MVAPICH2 2.0b.

For testing purposes, we picked 4 large social network graphs from the SNAP dataset to perform the tests at scale. The properties of the graphs are summarized in Table 1. 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 1. 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.

Name	$\|V\|$	$\|E\|$	# updates	time(s)
`web-NotreDame`	325,729	1,090,008	399,420	3.29
`amazon0601`	403,394	2,443,308	1,548,288	33.16
`web-Google`	916,428	4,321,958	2,527,088	71.20
`soc-pokec`	1,632,804	30,622,464	4,924,759	816.73

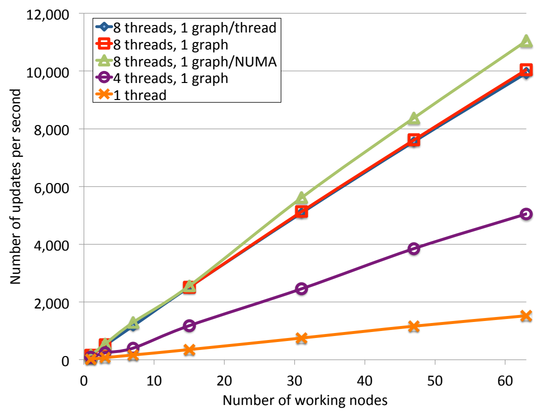
Table 1: Properties of the graphs we used in the experiments and execution time on a 64 node cluster.

All the results presented in this section are extracted from a single run of Streamer with proper parameters. As our preliminary results show, 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.

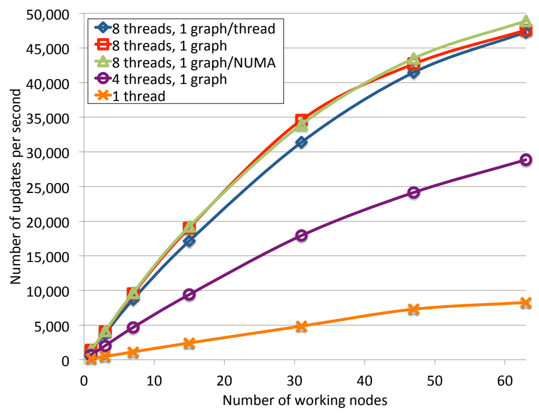
To give an idea of the actual amount of computation, in the last column of Table 1, we report the time Streamer spends to update the CC scores upon 50 edge insertions by using all 63 worker nodes. We present the parallel time and not the sequential time for two reasons: (1) Our framework is never really 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 1 do not contain the initialization time. That is the time measurement starts once Streamer is idle, waiting to receive Streaming Events.

4.1. Basic performance results

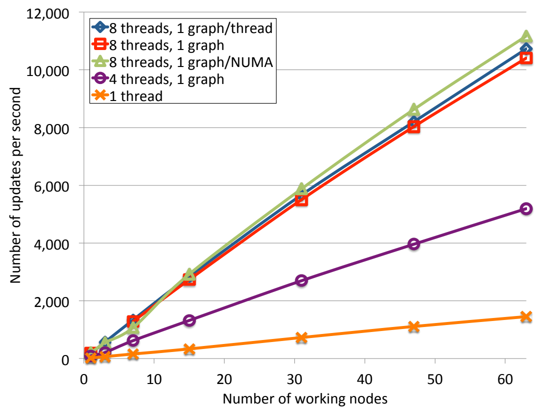
Figure 9 shows the performance and scalability of the system in different configurations with a single *StreamingMaster* and *Aggregator*. The performance is expressed in number of updates per second. The framework obtains up to $11,000$ updates/sec on `amazon0601` and `web-Google`, $49,000$ updates/sec 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 (504 *Executor* threads in total) compared to the single node-single thread configuration (the incremental centrality computation on



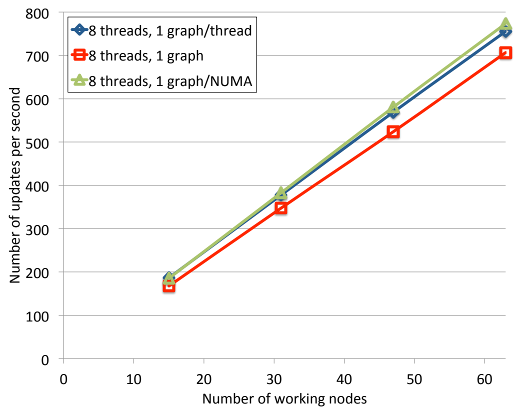
(a) `amazon0601`



(b) `web-NotreDame`



(c) `web-Google`



(d) `soc-pokec`

Figure 9: 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.

`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.

Let us first evaluate the performance obtained under different node-level configurations. Table 2 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 a speedup of 6.5 or more. Overall, having multiple cores is

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

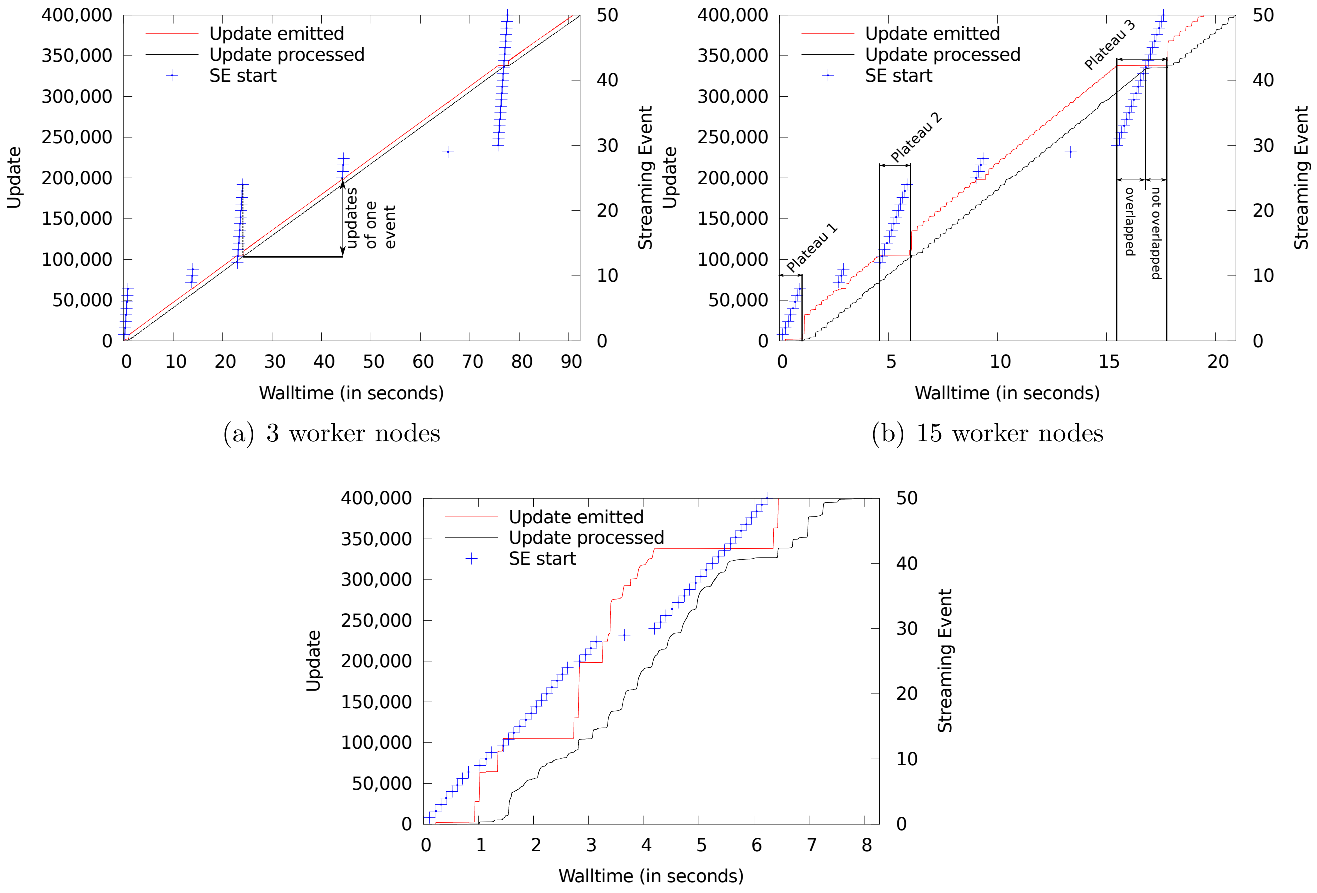
Table 2: The performance of STREAMER with 31 worker nodes and different node-level configurations normalized to 1 thread case (performance on `soc-pokec` is normalized to 8 threads, 1 graph/thread). The last column is the advantage of Shared Memory awareness (ratio of columns 5 and 3).

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. **UVC: Erik will fix this :)** 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.

4.2. Execution-log analysis

Here we discuss the impact of pipelined parallelism and the sub-linear speedup achieved on `web-NotreDame`. In Figure 10, 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. 10(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 *StreamingMaster* 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



(a) 3 worker nodes

(b) 15 worker nodes

(c) 63 worker nodes

Figure 10: 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).

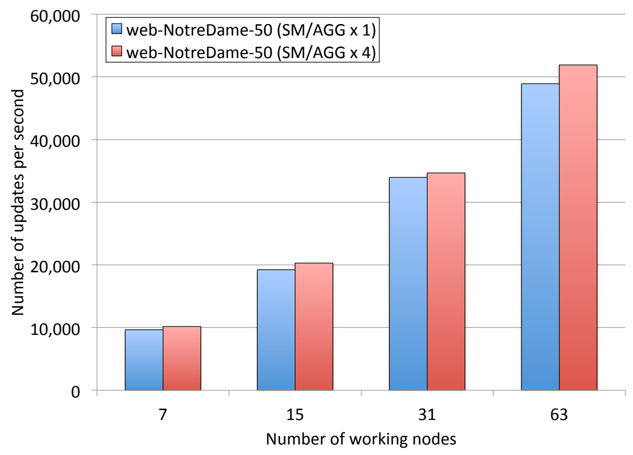
first line). In the first plot, we can see that 6 out of 50 Streaming Events (the ticks at the end of each partial tick-lines) incurred significantly much more updates than the others. While these events are being processed, the two lines stay straight and parallel, because in DataCutter, writing to 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. 10(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 *StreamingMaster* spends all its time by filtering the work for these Streaming Events.

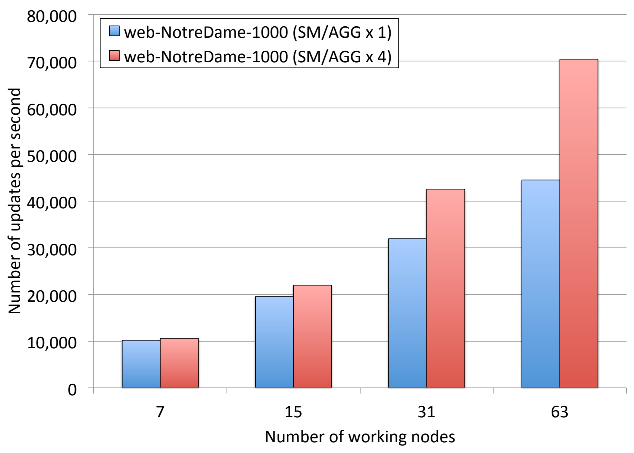
The second plateau around time 5 seconds of the execution log with 15 worker nodes lasts 1.2 secs, and less than 100 updates are sent during that interval. However, as the plot shows, the worker nodes do not run out of work and process more than $25,000$ updates during the plateau. This is possible because 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. 9(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. 10(c)) presents another picture. With the increase on the workers' processing power, a single *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.

We used the techniques described in Sections 3.3 and 3.4 (Figs. 7 and 8) to replicate the *StreamingMaster* and *Aggregator* filters, respectively, and obtain a better performance when these filters becomes bottleneck throughout the incremental closeness centrality computation. The results on the `web-NotreDame` graph are given for 50 and $1,000$ Streaming Events in Figure 11. As the figure shows, using four *StreamingMaster* and *Aggregator* filters instead



(a) 50 edge insertions on web-NotreDame



(b) 1,000 edge insertions on web-NotreDame

Figure 11: Parallelizing streaming master: the number of updates per second for `web-NotreDame` with 50 and 1,000 streaming events, respectively. The best node configuration from Figure 9, i.e., 8 threads, 1 graph/NUMA, is used for both cases.

of one yields around 6% improvement for 50 Streaming Events when 63 working nodes in the cluster are fully utilized. This small improvement is due to lack of sufficient number of Streaming Events which generates a large amount of updates (see Figure 10). Hence, even with a large number of *StreamingMaster* and *Aggregator* filters, due to the load balancing problem on these filters, one cannot improve the performance more with 50 Streaming Events by just replicating them. Fortunately, in practice this number is usually much higher. In Figure 11(b), we repeated the same experiment for 1,000 Streaming Events. As the figure shows, the performance significantly increases when the filters are replicated. Furthermore, the percentage of the improvement increases when more number of nodes are used and reaches to 58% with 63 working nodes. This is expected since, with more number of nodes for the *Executor* filter, the time spent for *StreamingMaster* and *Aggregator* becomes more important. When applied on the other graphs, going from one *StreamingMaster* and *Aggregator* to four have not yield significant improvements since these components were not bottlenecks. Therefore, we omited those results here.

4.3. Plug-and-play filters: Vectorization

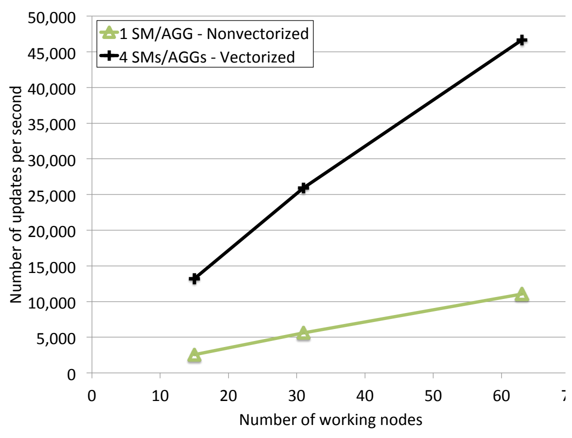
As stated above, thanks to filter-stream programming model, different filter implementations and various hardware such as GPUs can be used easily and efficiently if desired. Here, we show that using the SpMM-based approach described in Section 2.5, one can modify the *ComputeCC* filter in Figure 5 (or the *Executor* filters in Figure 6) to increase the speedup. For this experiment, we changed the filter in a way that 32 BFSs from different sources are executed in parallel to update the corresponding closeness centrality values. The results of the experiments with 15, 31, and 63 working nodes are shown in Figure 12. Using a different kernel with vectorization (and coupled with multiple *StreamingMaster* and *Aggregator*) improves the performance of the non-vectorized version by a factor ranging from 2.2 to 9.3 depending on the graph and number of working nodes.

4.4. Summary of the experimental results

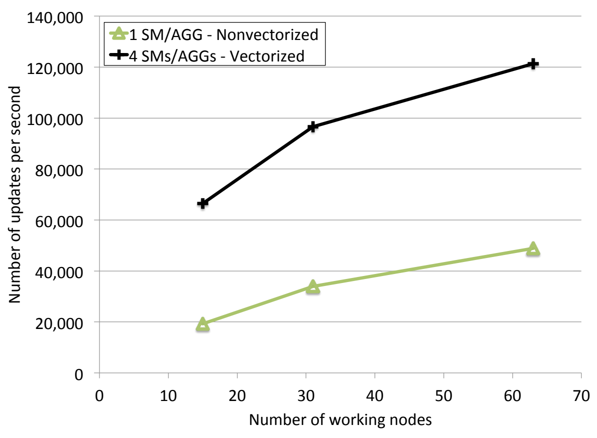
The experiments we conducted shows that Streamer can scale up and efficiently utilize our entire experimental cluster. By taking the hierarchical composition of the architecture into account (64 nodes, 2 processors per node, 4 cores per processor) and not considering it as a regular distributed machine (a 512-processor MPI cluster), we enabled processing of larger graphs and obtained 10% additional improvement. Furthermore, the pipelined parallelism proved to be extremely necessary while using a large amount of nodes in a concurrent fashion.

Replicating the *ComputeCC* filter leads to significant speedup. Yet, the bottleneck eventually becomes the filters that cannot be replicated automatically. For such filters, that the ordering of the messages is important, we can substitute an alternative filter architecture to alleviate the bottleneck and make the whole analysis pipeline highly parallel.

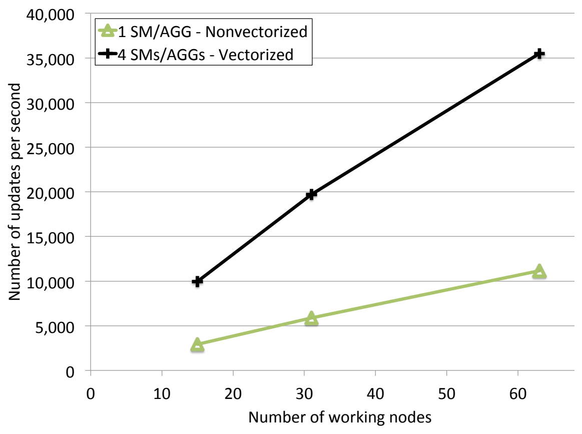
The flexibility of the filter-stream programming model allows to easily substitute a component of the application by an alternative implementation. For instance, one can use modern vectorization techniques to improve the performance by a significant factor. Similarly, one could have alternative implementation which use different type of hardware such as accelerators.



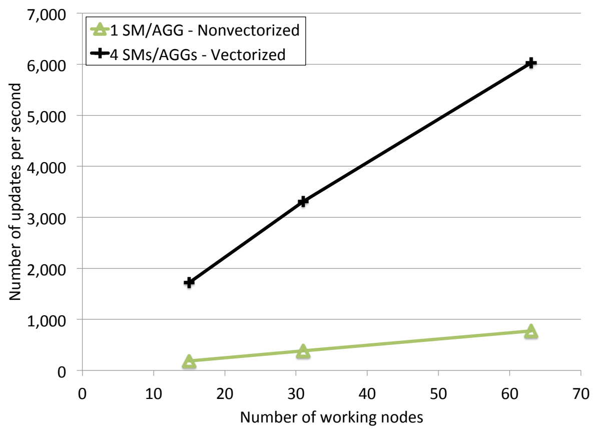
(a) `amazon0601`



(b) `web-NotreDame`



(c) `web-Google`



(d) `soc-pokec`

Figure 12: Vectorization: the performance is expressed in the number of updates per second. The best worker-node configuration, "8 threads, 1 graph/NUMA", is used for the experiments.

5. 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 almost linear speedup for the three out of four graphs we used for the experiments.

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. We had this problem for one of our graphs, and could increase the performance further by replicating the *StreamingMaster* and *Aggregator* filters which have almost negligible overhead in a sequential execution. We also used techniques such as vectorization for a much faster incremental centrality computation which shows that one can use different filter implementation or even a better hardware to increase the performance of Streamer.

Acknowledgments

This work was supported in parts by the NSF grant OCI-0904809, and the Defense Threat Reduction Agency grant HDTRA1-14-C-0007.

References

[1] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader. Scalable graph exploration on multicore processors. In *SuperComputing*, pages 1–11, 2010.

[2] M. Belgin, G. Back, and C. J. Ribbens. Pattern-based sparse matrix representation for memory-efficient SMVM kernels. In *Proc. of ICS*, pages 100–109, 2009.

[3] 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.

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

[5] A. Buluç and J. R. Gilbert. The combinatorial BLAS: Design, implementation, and applications. *International Journal of High Performance Computing Applications (IJHPCA)*, 2011.

[6] A. Buluç, S. Williams, L. Oliker, and J. Demmel. Reduced-bandwidth multithreaded algorithms for sparse matrix-vector multiplication. In *Proc. IPDPS*, 2011.

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

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

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

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

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

[12] 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.

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

[14] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart. Edge vs. node parallelism for graph centrality metrics. In *GPU Computing Gems: Jade Edition*. Morgan Kaufmann, 2011.

[15] 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.

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

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

[18] 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.

[19] X. Liu, M. Smelyanskiy, E. Chow, and P. Dubey. Efficient sparse matrix-vector multiplication on x86-based many-core processors. In *Proceedings of the 27th international ACM conference on International conference on supercomputing*, ICS '13, 2013.

[20] 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.

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

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

[23] P. Pande and D. A. Bader. Computing betweenness centrality for small world networks on a GPU. In *15th Annual High Performance Embedded Computing Workshop (HPEC)*, 2011.

[24] 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.

[25] 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.

[26] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek. Hardware/software vectorization for closeness centrality on multi-/many-core architectures. In *28th International Parallel and Distributed Processing Symposium Workshops, Workshop on Multithreaded Architectures and Applications (MTAAP)*, May 2014.

[27] A. E. Sarıyüce, E. Saule, K. Kaya, and Ümit V. Çatalyürek. Shattering and compressing networks for betweenness centrality. In *Proc. of SDM*, 2013.

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

[29] R. Vuduc, J. Demmel, and K. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC 2005, J. of Physics: Conference Series*, 2005.