Hieroglyph: Locally-Sufficient Graph Processing via Compute-Sync-Merge

Xiaoen Ju
University of Michigan
jux@umich.edu

Hani Jamjoom
IBM T.J. Watson Research Center
jamjoom@us.ibm.com

Kang G. Shin
University of Michigan
kgshin@umich.edu

ABSTRACT

Despite their widespread adoption, large-scale graph processing systems do not fully decouple computation and communication, often yielding suboptimal performance. Locally-sufficient computation—computation that relies only on the graph state local to a computing host—can mitigate the effects of this coupling. In this paper, we present *Compute-Sync-Merge* (CSM), a new programming abstraction that achieves efficient locally-sufficient computation. CSM enforces local sufficiency at the programming abstraction level and enables the activation of vertex-centric computation on all vertex replicas, thus supporting vertex-cut partitioning. We demonstrate the simplicity of expressing several fundamental graph algorithms in CSM. Hieroglyph—our implementation of a graph processing system with CSM support—outperforms state of the art by up to 53x, with a median speedup of 3.5x and an average speedup of 6x across a wide range of datasets.

1. INTRODUCTION

Mainstream graph processing systems (such as Pregel [23], PowerGraph [12], and GraphX [13]) follow the bulk synchronous parallel (BSP) model [31] in which they iteratively and synchronously apply a vertex-centric algorithm on a graph. Inherent in their design is the tight coupling of computation and communication, where no vertex can proceed to the next iteration of computation until all vertices have been processed in the current iteration and graph states have been synchronized across all hosts. Especially for computationally-light graph algorithms, this coupling of computation and communication incurs significant performance penalty [35].

Intuitively, if graph processing systems can fully decouple computation from communication, they should achieve higher performance because they can better overlap communication and computation. We argue that fully decoupling computation from communication can be achieved; it requires (*i*) restricted access to only local state during computation and (*ii*) independence of inter-host communication from computation. We call the combination of both conditions *local sufficiency*. Conceptually, local sufficiency allows all vertices to always make progress without blocking on input from remote vertices (i.e., those residing on remote hosts). Since all vertices are executing in parallel, local sufficiency can introduce state inconsistency in two ways: (*i*) vertices might make progress using incomplete input, and (*ii*) replicated vertices might make progress—in parallel—on different sets of inputs. As we will show in the paper, resolving both types of inconsistencies can be done efficiently.

Local sufficiency is not efficiently supported by state of the art systems. Synchronous systems, by design, do not support local sufficiency due to their intrinsic computation-communication coupling. Even systems that implement asynchronous execution only partially achieve local sufficiency. For example, PowerGraph [12]'s asynchronous mode satisfies local sufficiency by distributed scheduling. If a vertex-centric function uses remote state, then it will not be marked as ready for execution until its remote input becomes locally available after state propagation. Thus, the function itself is not locally sufficient. Furthermore, the scheduling overhead can be substantial. GiraphUC [14] avoids such a cost by concentrating computation on master vertex replicas, efficiently supporting local sufficiency for edge-cut partitioning. But this approach does not support vertex-cut and thus cannot benefit from its balanced workload distribution.

Towards efficient support for local sufficiency, we set two design goals. The first goal is to activate vertex-centric computation on all vertex replicas, enabling each replica to independently update its local state. This relaxed consistency model would support vertex-cut and enable fast local state propagation without inter-host coordination. The second goal is to enforce local sufficiency at the programming abstraction level. This would eliminate any related coordination overhead at the system level. Additionally, any inconsistency would be resolved by user-defined functions, which are coordinated across all hosts to achieve globally-consistent state upon convergence.

Following these design choices, we introduce a new programming abstraction called *Compute-Sync-Merge* (CSM). The Compute abstraction (*Compute* for short) defines locally-sufficient computation, which is iteratively and independently applied to all vertex replicas on each host. Local sufficiency is enforced by the abstraction. *Compute* has access only to local input state. The Sync and Merge abstractions (referred to as *Sync* and *Merge* henceforth) coordinate the execution on all hosts. The former is in charge of state propagation and the latter is responsible for the merging of remote updates with local state. Together, they resolve the inconsistency caused by locally-sufficient computation.

We demonstrate the expressiveness and simplicity of CSM by implementing several widely-used single-phase algorithms, such as PageRank, single-source shortest path (SSSP), and weakly connected component. The CSM abstraction also provides a new dimension for designing efficient locally-sufficient multi-phase graph

SIGMETRICS '17 June 5–9, 2017, Urbana-Champaign, IL, USA

DOI: 10.475/123_4

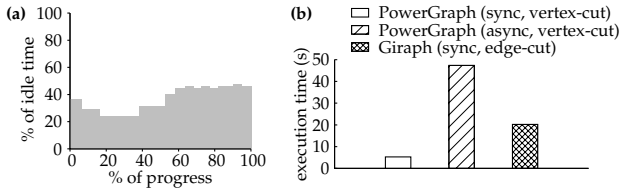

Figure 1: Deficiency when executing SSSP over a Twitter followers graph with PowerGraph and Giraph on 16 Amazon EC2 c3.8xlarge instances. (a) Instantaneous idle time (in gray) of PowerGraph. (b) Comparison of execution modes and graph partitioning.

algorithms. In general, multi-phase algorithms limit the performance gains of locally-sufficient computation due to global synchronization at phase boundaries [14]. CSM, however, enables the design of multi-phase algorithms in which (*i*) locally-sufficient computation freely proceeds beyond phase boundaries and (*ii*) conflicting state due to computation with local input is resolved in *Sync* and *Merge*. We exemplify such use of CSM with an efficient new design of a multi-phase maximal bipartite matching algorithm.

We have fully implemented *Hieroglyph*, a graph processing system supporting CSM on top of PowerLyra [7]. We extend PowerGraph's gather-apply-scatter (GAS) abstraction [12] to realize the CSM abstraction, augmenting the portability of GAS-based algorithms to CSM. Experiments with real-world graphs show that Hieroglyph consistently and significantly outperforms state of the art systems. It outperforms PowerGraph and PowerLyra by up to 22x and GiraphUC by up to 53x, achieving a median speedup of 3.5x and an average speedup of 6x among all algorithm-dataset combinations in our evaluation.

The contributions of this paper include:

- the introduction of the Compute-Sync-Merge (CSM) abstraction, enabling efficient local-sufficiency on vertex-cut partitioning,
- the design of several CSM algorithms to demonstrate the expressiveness, simplicity, and efficiency of the abstraction, and
- the implementation of Hieroglyph, a graph processing system supporting the CSM abstraction and significantly outperforming state of the art systems.

The remainder of the paper is as follows. Section 2 provides background information on state-of-the-art graph processing. We introduce the CSM abstraction in Section 3 and present Hieroglyph, a CSM-compliant system, in Section 4. Section 5 demonstrates Hieroglyph's superiority by comparing its performance with three state-of-the-art systems. Related work is discussed in Section 6. We then conclude the paper in Section 7.

2. BACKGROUND

Synchronous Model. In a synchronous model, vertex-centric computation proceeds in supersteps. Computation and communication iterations alternate. Such coupling can cause a significant performance penalty. Figure 1a shows the instantaneous idle time[1] due to communication when running SSSP on a Twitter followers graph with PowerGraph. Along the course of the execution, the idle time remains substantial, indicating considerable blocking of computation due to communication.

Asynchronous Model. Asynchronous execution [12] decouples computation and communication at the vertex boundary: per-vertex computation-communication tasks are no longer aligned by supersteps and can be independently performed, improving the computation-communication interleaving. Such cross-vertex computation-communication decoupling partially satisfies local sufficiency. For a given vertex, however, the coupling still exists, albeit hidden behind the scheduling of the processing system. Regarding performance, the scheduling overhead, along with the communication deficiency due to the lack of message batching and increased locking overhead, causes asynchronous execution to underperform synchronous execution for several common algorithms, despite the former's faster convergence [21]. For example, Figure 1b shows that PowerGraph's synchronous execution significantly outperforms its asynchronous mode, when running SSSP over the Twitter graph.

GiraphUC [14] proposes barrierless asynchronous model, achieving local sufficiency with the observations that (*i*) inter-host communication is much more expensive than intra-host communication and (*ii*) computation can proceed with partial input state propagated via intra-host communication. Remote input state is consumed when it becomes available. GiraphUC supports local sufficiency over *edge-cut* partitioning. That is, workload related to a vertex—including vertex state update and message passing—must be conducted by only one host. Prior work [7, 12] shows that edge-cut partitioning leads to more skewed computation workload and larger memory footprint compared to *vertex-cut*, the latter capable of distributing per-vertex workload onto multiple hosts, each holding a replica of that vertex. Figure 1b compares the performance of PowerGraph synchronous execution with Giraph [1], an open-source implementation of Pregel. Using vertex-cut to produce balanced graph partitions and evenly distribute computation workload, PowerGraph yields a 4x speedup with respect to Giraph, the latter supporting only edge-cut.[2] Given the substantial performance discrepancy between vertex-cut and edge-cut, enabling local sufficiency only for edge-cut would miss an important opportunity for performance improvement.

3. COMPUTE-SYNC-MERGE

In this section, we discuss the Compute-Sync-Merge (CSM) abstraction, including its challenge, design, workflow, consistency model, and expressiveness.

3.1 Challenge

As mentioned in Section 1, local sufficiency requires (*i*) restricting access to only local state available during vertex-centric computation and (*ii*) independence of inter-host communication from local computation. Most existing abstractions cannot fully support local sufficiency, because of the use of remote input state in vertex-centric computation. Such usage is expressed, for example, in the form of message exchanges in Pregel, vertex state synchronization in Distributed GraphLab [22] and Cyclops' distributed immutable view [6], and distributed gathering/scattering in PowerGraph. GiraphUC achieves local sufficiency for edge-cut, but lacks support for vertex-cut.

Supporting local sufficiency on vertex-cut is challenging because of the partial distribution of vertex-centric computation. Specifically, vertex-cut distributes per-vertex computation workload onto multiple hosts, each maintaining a replica of that vertex. In order to

[1]The instantaneous idle time is the percentage of execution time that is idle at time t. That is, if in a time interval $(t, t + dt)$, the total idle time is $p\,dt$, then the instantaneous idle time at time t is p.

[2]Differences in systems implementation also contribute to the performance discrepancy.

Table 1: CSM Abstraction

compute(D_u, N_u) $\rightarrow$ (D_u^{new}, N_u^{new})
sync(D_u) $\rightarrow$ M_u
merge(D_u, N_u, M_u) $\rightarrow$ (D_u^{new}, N_u^{new})

Table 2: Notation

Symbol	Semantics
D_u	data of vertex u
$D_{(u,v)}$	data of edge $u \rightarrow v$
N_u	neighboring states of u, i.e., $\bigcup_{v \in \text{nghr}(u)} \{D_v, D_{(u,v)}\}$
M_u	final sync data for vertex u
I_u^{local}	local sync data for vertex u
I_u^{mirror}	sum of sync data from mirrors of u
I_u^{master}	interim sync data for u at master

guarantee vertex state consistency, however, any vertex state update is restricted to the host holding the master replica. Consequently, per-vertex computation is partially distributed: only non-state-update functions are processed by all replicas. Thus comes the need to coordinate the computation across multiple replicas of a vertex, for example, by properly aligning the computation stages in which each replica resides. This, in turn, leads to the coupling of computation and communication. Such coupling needs to be resolved to achieve local sufficiency on vertex-cut. In contrast, edge-cut concentrates computation related to a vertex to only one host. The challenge imposed by partial distribution of vertex-centric computation thus does not apply to edge-cut.

3.2 Abstraction and Workflow

To achieve local sufficiency in a vertex-cut abstraction, we make two design decisions. First, for vertex-centric computation on each host, we confine the input scope to local state at the programming abstraction level. Such enforcement elimintates the need for sophisticated local-sufficiency-related coordination at the system level. Second, we activate vertex-centric computation on all vertex replicas and, if such computation consists of multiple stages (e.g., gather-apply-scatter), we activate all stages on all vertex replicas and enable autonomous stage transition without inter-host coordination. This allows computation to proceed at full speed over a vertex-cut partitioned graph. Synchronization of vertex state and merging of local and remote states are necessary in such a design to achieve consistency of the final converged graph state across all hosts. We expose them at the programming abstraction level, enabling flexible inconsistency resolution.

Following this reasoning, we design Compute-Sync-Merge (CSM) (cf. Table 1). We discuss CSM along its workflow (cf. Figure 2). A CSM-compliant system maintains two groups of worker threads, one for computation and the other for communication. Locally-sufficient computation progresses in iterations, independently on each host. For an iteration, the computation workers invoke a *Compute* function on all active vertices. A vertex is activated for the next iteration, if it receives messages as a result of the current iteration of computation. The next iteration starts immediately after the completion of the current iteration, without inter-host communication.

The communication workers run in parallel with the computation workers, synchronizing local updates across all vertex replicas with a *Sync* function. State synchronization also progresses in iterations.

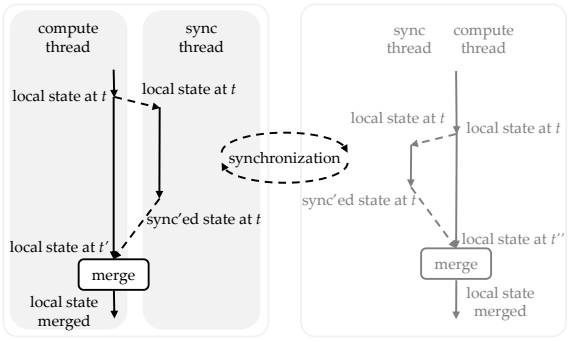

Figure 2: Interaction of computation, synchronization, and merging in CSM workflow, illustrated with two hosts (placed left and right). One compute thread and one synchronization thread are shown for each host. Logical time progresses downward.

Contrary to computation iterations, which are locally maintained, synchronization iterations are aligned across all hosts by global barriers. Upon completion of a synchronization iteration, states propagated from remote hosts are merged with the local states used for locally-sufficient computation via a *Merge* function.

3.3 Consistency and Expressiveness

To support local sufficiency on vertex-cut, CSM employs a relaxed consistency model for vertex-centric computation. Specifically, the state of a local replica can be updated independently to other replicas of the same vertex. No ordering constraint is imposed on such updates. Resolving the inconsistency caused by parallel updates, as well as the achievement of eventual consistency upon task completion, depends on the collaboration of *Compute*, *Sync*, and *Merge*.

An algorithm can be expressed by CSM if it can tolerate transient inconsistency. It requires the following conditions:

- There exists an inconsistency resolution procedure that synchronizes inconsistent state across multiple vertex replicas due to locally-sufficient computation. Thus, inconsistency across vertex replicas remains transient.
- Such a procedure preserves forward progress. Otherwise, the execution may endlessly switch between locally-sufficient computation and inconsistency resolution, affecting convergence.
- The resolution result—the consistent state of a vertex—corrects the state of other vertices generated by using the previously inconsistent vertex state as input.

Several common graph algorithms, such as SSSP, weakly connected component, PageRank, maximal bipartite matching, betweenness centrality, and approximate diameter, are tolerant of transient state inconsistency and can be easily expressed by CSM.

CSM's expressiveness of inconsistency resolution can lead to better performance, if the gain from local sufficiency outweighs the cost incurred by inconsistency resolution. Such is the case for the four algorithms studied in this paper. As expected, with such expressiveness comes an increase in algorithm complexity. We evaluate both aspects in Section 5.

Algorithms intolerant of local inconsistency can also be expressed in CSM, albeit without the potential performance gain from local sufficiency. This is because stronger consistency can be flexibly reintroduced to the workflow via the three functions in CSM. For example, GAS can be expressed in CSM by (*i*) reducing the collec-

Table 3: Hieroglyph's Implementation of CSM

Compute
gather(D_u, $D_{(u,v)}$, D_v) $\rightarrow acc$
sum(acc left, acc right) $\rightarrow acc$
apply(D_u, acc) $\rightarrow D_u^{new}$
scatter(D_u^{new}, $D_{(u,v)}$, D_v) $\rightarrow D_{(u,v)}^{new}$
Sync
sync_switch(D_u) $\rightarrow I_u^{local}$
sync_sum(I_u^{local} left, I_u^{local} right) $\rightarrow I_u^{local}$
sync_apply(I_u^{local}, I_u^{mirror}) $\rightarrow I_u^{master}$
sync_commit(I_u^{local}, I_u^{master}) $\rightarrow M_u$
Merge
merge_apply(D_u, M_u) $\rightarrow D_u^{new}$
merge_scatter(D_u^{new}, $D_{(u,v)}$, D_v, M_u) $\rightarrow D_{(u,v)}^{new}$

tion of local gather states to *Compute*, (*ii*) reducing the propagation of gather states, the generation of the updated master state in the apply function, and the synchronization of the master state, to *Sync*, and (*iii*) reducing the vertex state update in the apply function, as well as local scatter, to *Merge*. GAS algorithms can thus be converted to their corresponding CSM versions.

4. Hieroglyph

We have fully implemented a CSM-compliant graph processing system called Hieroglyph. It is implemented as an integrated component of PowerLyra [7], which is, in turn, built on PowerGraph [12]. Hieroglyph extends PowerLyra and PowerGraph's vertex-centric abstraction to support CSM. It implements the CSM workflow in a standalone processing engine, parallel to existing ones in PowerGraph and PowerLyra. It also extends PowerGraph's graph analytics toolkit with algorithms implemented with the CSM abstraction.

4.1 Implementation of CSM in Hieroglyph

Hieroglyph decomposes the CSM abstraction into several primitive functions (cf. Table 3). We detail this implementation along Hieroglyph's workflow.

Compute. In each locally-sufficient computation iteration (cf. Algorithm 1, Line 3), the computation workers iterate through all active local vertices. Vertex-centric computation is implemented in a GAS style. The reuse of the GAS abstraction in CSM's *Compute* increases the portability of existing GAS-based algorithms to CSM. For each vertex, information regarding its neighbor vertices and edges is accumulated through a *gather-sum* function pair or via a *sum* over messages sent by a previous scatter stage. Such information is then used to update the vertex state via an *apply* function. A *scatter* function concludes the vertex-centric computation by updating neighbor edges according to the new vertex state. Hieroglyph's Compute differs from PowerGraph's GAS in that (*i*) all of its three stages are executed on all active vertex replicas and (*ii*) its stage transition is autonomous, without inter-host coordination.

Sync. After an iteration of computation, the computation workers resume the communication workers to perform metadata synchronization (cf. Algorithm 1, Line 6). They then continue with locally-sufficient computation. The goal of metadata synchronization is to achieve consensus among all hosts regarding the set of to-be-synchronized vertices. At the end of the metadata synchronization, if no progress can be made by any host since the last synchronization (cf. Algorithm 1, Line 12), then the computation has completed. Otherwise, computation workers invoke a *sync_switch* function on to-be-synchronized vertices. The purpose of *sync_switch* is to create a standalone copy of the subset of vertex state used for communication, so that subsequent computation can freely proceed, updating vertex state without conflicting with communication. [3] After switching state, the computation workers resume locally-sufficient computation tasks, delegating vertex state synchronization to the communication workers.

The vertex state synchronization is divided into three stages, separated by global barriers (cf. Algorithm 3). In the first stage, all mirror replicas (I_u^{mirror}) are sent to their corresponding master hosts. In the second stage, each host generates intermediate master copy (I_u^{master}) by combining master replicas with received mirror state in a *sync_apply* function and distributes the master copy to the corresponding mirroring hosts. In the third stage, each host creates the final merging state (M_u) for all vertices participating in the current synchronization by combining the local synchronization state and the master copy in a *sync_commit* function. [4]

Algorithm 1 Overall Execution Flow

```
1:  /* executed by compute workers */
2:  while true do
3:      compute an iteration
4:      if sync_worker_state == INACTIVE then
5:          sync_worker_state ← META
6:          resume sync workers for metadata sync
7:      else
8:          if sync_done == false then
9:              continue /* to next iteration of computation */
10:         sync_done ← false
11:         if sync_worker_state == META then
12:             if has_updated_vertices == 0 then
13:                 /* all hosts have converged */
14:                 terminate computation
15:             else
16:                 sync_switch /* prepare sync data */
17:                 sync_worker_state ← DATA
18:                 resume sync workers for data sync (cf. Algo. 3)
19:         else
20:             sync_worker_state ← INACTIVE
21:             merge
```

Algorithm 2 Execution Flow: Metadata Synchronization

```
1: /* executed by communication workers */
2: has_updated_vertices ← local_updated_vids.count()
3: ∑^{all hosts} has_updated_vertices
4: /* exclude single-replica (i.e., internal) vertices */
5: updated_vids ← local_updated_vids − internal_vids
6: sync updated_vids across all hosts
7: sync_done ← true
```

[3] Optimization related to benign contention is discussed in Section 4.

[4] Note that, when vertex state is updated independently by each replica, there is no distinction between master and mirror in the vertex-centric computation phase. As a result, synchronization cannot be simply reduced to overwriting mirrors with the master state and needs to be exposed at the programming abstraction. This explains the introduction of *sync_apply* and *sync_commit* in *Sync*.

Algorithm 3 Execution Flow: Data Synchronization

```
1: /* executed by communication workers */
2: for u in updated_vids do
3:    if u is not master copy then
4:       send I_u^local to master
5: global barrier
6: for u in updated_vids do
7:    if u is master copy then
8:       I_u^master ← sync_apply(I_u^local, I_u^mirror)
9:       send I_u^master to mirror hosts
10: global barrier
11: for u in updated_vids do
12:    M_u ← sync_commit(I_u^local, I_u^master)
13: sync_done ← true
```

Algorithm 4 SSSP in CSM

```
Compute
gather(D_u, D_(u,v), D_v): no-op
sum(a, b): return min(a, b) /* message combiner */
apply(D_u, acc): D_u = min(D_u, acc)
scatter(D_u, D_(u,v), D_v): /* to out neighbors */
   if changed(D_u) and (D_u + D_(u,v) > D_v) then
      send_msg(v, D_u + D_(u,v))

Sync
sync_switch(D_u): I_u^local = D_u
sync_sum(a, b): return min(a, b)
sync_apply(I_u^local, I_u^mirror): I_u^master = min(I_u^local, I_u^mirror)
sync_commit(I_u^local, I_u^master): M_u = I_u^master

Merge
merge_apply(D_u, M_u): D_u = min(D_u, M_u)
merge_scatter(D_u, D_(u,v), D_v, M_u): /* to out neighbors */
   if changed(D_u) and (D_u + D_(u,v) > D_v) then
      send_msg(v, D_u + D_(u,v))
```

Merge. Upon communication completion, the computation workers enter the merging stage (cf. Algorithm 1, Line 21). They first use a *merge_apply* function to merge remote state with the local counterpart. Another *merge_scatter* function is then invoked to update neighbor edges according to the newly merged state. [5]

4.2 CSM Algorithm Design: Case Studies

We exemplify single- and multi-phase CSM algorithm designs with SSSP and bipartite matching.

SSSP. For SSSP (cf. Algorithm 4), *Compute* is expressed similarly to its counterpart in common vertex-centric abstractions [12]. If the shortest distance of a vertex changes due to *apply*, it notifies its neighbors in *scatter*. Messages are combined with the *min* operator, as shown in *sum*.

Sync involves the propagation of the minimum shortest distance among all replicas. Specifically, *sync_switch* first makes a copy of the local vertex state, which is then propagated from mirror replicas to the master. The shortest distances from mirror replicas are combined with the *min* operator in *sync_sum*. The intermediate master value is the minimum of the shortest distance of the master replica and that of the received mirror replicas. It is then broadcasted to all mirror replicas. *Sync_commit* establishes the intermediate master value as the final merge value.

As for *Merge*, *merge_apply* and *merge_scatter* have identical functionality to their counterparts in *Compute*. Thus, if the local vertex state has a shortest distance that is no larger than the merge value, then the merge value is ignored. This is the case when the current replica contributes the minimum shortest distance to the previous synchronization iteration. It can also happen because locally-sufficient computation further advances the vertex state during the previous synchronization. When merging, the current local vertex state may have become smaller than the minimum shortest distance of all replicas in the previous synchronization. Otherwise, the current local state is larger than the merge value and is overwritten by the latter in *merge_apply*. Such an update is then propagated to local neighbors in *merge_scatter*.

In summary, SSSP demonstrates the simplicity and elegance of algorithm design using CSM. *Compute* handles locally-sufficient computation and is identical to PowerGraph's GAS implementation, facilitating design reusing. In addition, *Sync* and *Merge* rely on the same message combining and vertex updating logic in *Compute*'s *sum* and *apply*, leading to simple inconsistency resolution.

Regarding transient inconsistency resolution, SSSP's *Sync* and *Merge* are designed so that all replicas of a vertex obtain a consistent view, that is, the minimum of local shortest distances. It also preserves forward progress: the resolved state is aligned to the local state that has advanced the most in locally-sufficient computation. In addition, the propagation of the resolution result in *Merge*—the new minimum—leads to the correction of neighbor vertex states, if they are generated using a previously inconsistent state.

The CSM implementation of SSSP is guaranteed to converge to the correct final graph state. This is because, if a destination vertex v_d is unreachable from the source vertex v_s, then v_d converges upon algorithm initialization (i.e., ∞). If v_d is reachable, then let n denote the minimum number of vertices along the shortest path(s) from v_s to v_d (inclusive). v_d enters the final converged state within $n-1$ iterations of synchronization: after at most i iterations, vertices i steps away from v_s receive the correct final states, propagated via *scatter* or *merge_scatter*.

Bipartite Matching. Multi-phase algorithms impose new challenges to locally-sufficient computation [14]. Due to the different functionality across computation phases, global synchronization is generally required to guarantee the correctness of execution. In addition, messages sent from a phase to be processed by the subsequent phase must be hidden until phase switching. Otherwise, they will be combined with messages targeting at the current phase, producing erroneous results. As a result, locally-sufficient computation is applied only phase-by-phase in GiraphUC, reducing the performance gain.

CSM, in contrast, enables locally-sufficient computation to freely proceed across phase boundaries, when the following two conditions are satisfied. First, progress can be made *across* phases based only on local state. Second, inconsistent vertex state as a result of cross-phase locally-sufficient computation can be fixed without affecting the correctness of the final converged graph state.

We demonstrate CSM's potential in multi-phase algorithm design with our implementation of the bipartite matching algorithm. We adapt the four-phase matching algorithm in Pregel [23] for CSM's *Compute*. When applying it to locally-sufficient computation with no inter-host coordination, that algorithm produces a maximal matching on each host. Inconsistent final state may oc-

[5] Given that local vertex state may advance further during concurrent synchronization, simply overwriting the local state with the remote state may negate the progress of locally-sufficient computation. As a result, how to merge local and remote state also needs to be exposed at the programming abstraction, thus justifying the introduction of *merge_apply* and *merge_scatter* in *Merge*.

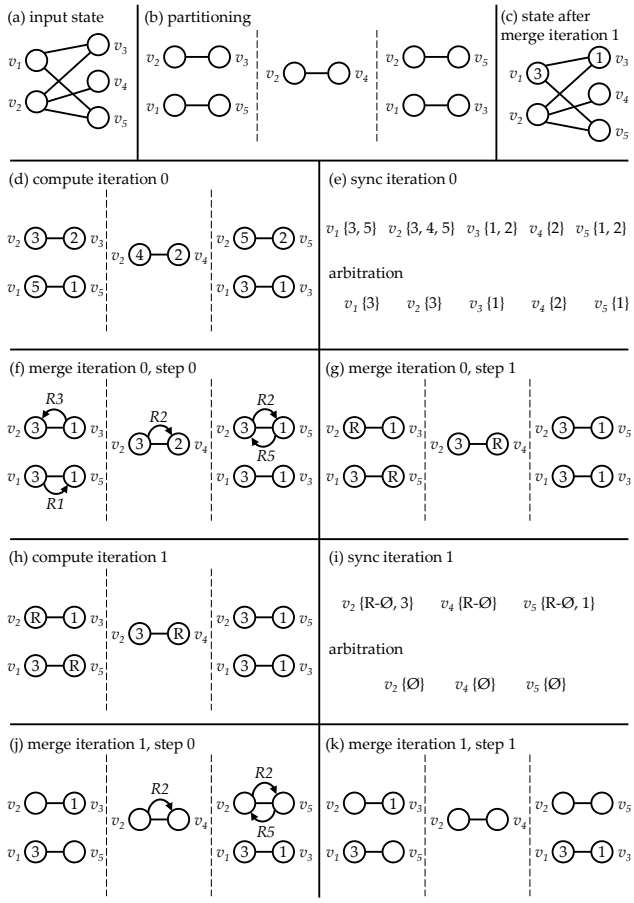

Figure 3: Illustration of bipartite matching in CSM. Each circle represents a vertex. The number inside a circle represents the id of the vertex with which the center vertex (represented by the circle) currently matches. A circled "R" represents the revoked state. An arrow associated with "R{vid}" represents a local revocation message with *vid* as its payload.

cur, nevertheless, in the form of vertices having different matching states across multiple replicas.

Such inconsistency is resolved in *Sync* and *Merge*. In a nutshell, *Sync* propagates local matching decisions to the master replicas of each vertex, which then use an arbitration function to resolve conflicting decisions. Such a decision is then distributed to all replicas of a vertex and is used as its merge value. *Merge* first corrects local decisions. Then, for each replica of which the matching decision is corrected, *Merge* notifies its previous partner vertex with a revocation message. This enables the previous partner vertex to enter a revoked state, become unmatched, and activate itself for the next iteration of computation. The proof of correctness of this implementation is presented in the appendix.

We use an example to describe bipartite matching in CSM, shown in Figure 3. The input graph (cf. Figure 3a) has five vertices, two on the left and three on the right, with five edges connecting them. Suppose the graph is partitioned onto three hosts (cf. Figure 3b). We follow the execution of the algorithm step by step until the completion of the second iteration of synchronization and merging (shown in Figures 3d–3k), reaching the state shown in Figure 3c. Note that, for bipartite matching, CSM confines the resumption of communication workers to the start of the four-phase matching cycles. The semantics of *one iteration of computation* in CSM's overall execution flow (cf. Algorithm 1, Line 3) thus refers to a cycle of four-phase matching. Intermediate state within a cycle is hidden from communication workers, simplifying the synchronization.

In the first iteration of computation, matching is performed locally (cf. Figure 3d). Local state is then synchronized (cf. Figure 3e), with the master replica of each vertex accumulating all local matching decisions. Each master replica then uses an arbitration function to resolve conflicting decisions. In our current implementation, each master replica selects the matching decision with the minimum matching vertex id. Such decision is then distributed to all replicas of a vertex and is used as its merge value.

Merge is separated into two steps in bipartite matching. In the first step, merge value obtained from synchronization is used to correct local decisions (cf. Figure 3f). When a local decision is corrected, it also sends a revocation message to its previous partner. For example, on the left host, after v_3's local decision of matching with v_2 is overwritten by the merge value v_1, v_3 sends a message to v_2, notifying its revocation of its previous matching decision. In the second step (cf. Figure 3g), each vertex receiving revocation messages checks the relevance of those messages. That is, whether the payload of a revocation message matches the current matching decision of the receiving vertex. Irrelevant revocation messages stem from global state synchronization: upon receipt of a revocation message, the vertex's state may have been updated to its globally consistent value. Since the receiving vertex has rematched to a vertex different than the sending vertex, the sender's revocation of a previous matching becomes irrelevant and is ignored. Such are the cases for v_2 and v_5 on the right host, each receiving an irrelevant revocation message from the other. When a relevant revocation message arrives (e.g., v_2 on the left host), the receiving vertex sets its state to revoked, becomes unmatched, and then activates itself for the next iteration of computation.[6]

The second iteration of computation progresses as the first one (cf. Figure 3h). Since all revoked vertices have their local neighbors in a matched state, no process can be made. In the subsequent synchronization (cf. Figure 3i), local revocations from the previous merging phase are propagated globally. The arbitration logic guarantees that revocation decisions overwrite matched state. For example, regarding v_2, the unmatched state (represented by $\varnothing$) prevails after a message combination with a matching decision of v_3.

The second iteration of merging proceeds as the first one (cf. Figures 3j and 3k), applying synchronized state and propagating local revocations in the first step and performing or ignoring revocations in the second step. After the merging, the graph reaches the state as shown in Figure 3c.

4.3 Discussion

Termination Condition. In Hieroglyph, the termination condition is checked against the total number of vertices updated across all hosts since the last synchronization (cf. Algorithm 1, Line 12). An algorithm completes when this number becomes zero.[7] In contrast, in the synchronous mode of PowerGraph and PowerLyra, this condition is checked at the superstep boundaries, owing to the tight coupling between computation and communication. If no progress

[6] In addition, a revoked right vertex also needs to activates all its unmatched left neighbors.

[7] This statement holds for single-phase algorithms. It also holds for multi-phase algorithms whose CSM implementations do not require global synchronization, such as our bipartite matching algorithm. For multi-phase algorithms mandating global synchronization at the phase-switching boundaries, this condition marks the end of a phase.

can be made after an iteration of computation followed by an iteration of communication, then the execution terminates.

The correctness of Hieroglyph's termination condition derives from the following patterns in the workflow (cf. Algorithm 1): (*i*) one iteration of computation (Line 3) is performed after an iteration of merging (Line 21) and before the termination condition checking (Line 12), and (*ii*) the termination checking, in turn, precedes the subsequent iteration of synchronization (Line 18). If no progress can be made in any host since the last synchronization, after incorporating the remote states in the merging stage and a subsequent iteration of locally-sufficient computation, then no progress is possible. Convergence over the entire graph has thus been achieved.

Computation-Communication Interleaving. In Hieroglyph, to achieve computation-communication decoupling, we use two groups of worker threads: one for computation and the other for communication. PowerGraph and PowerLyra, in contrast, rely on one group of threads to perform both computation and communication.[8]

The two groups of worker threads in Hieroglyph are of an equal size, both equating the number of cores on a computing host. Hieroglyph supports fine-tuning of computation-communication interleaving. On the one hand, in order to expedite the integration of remote vertex state updates into locally-sufficient computation, the computation workers separate vertices into chunks and perform computation one chunk at a time, yielding to the communication workers at the chunk boundaries and thus improving their interleaving.[9] On the other hand, Hieroglyph can be configured to enforce algorithm-specific restrictions on computation-communication interleaving. Our bipartite matching algorithm, for instance, confines the interaction between the two groups of worker threads only to the boundaries of a four-phase computation cycle. It ensures that local matching decisions, instead of intermediate states during locally-sufficient multi-phase computation, are used for synchronization.

Deferred Switching. Deferred switching refers to postponing the preparation of the synchronization state from the beginning of the synchronization to when the state is accessed by the communication workers. Specifically, it delegates the invocation of *sync_switch* from the computation worker (cf. Algorithm 3, Line 16) to the communication workers, before the sending of I_u^{local} in the case of a mirror replica (cf. Algorithm 3, Line 4) and before the invocation of *sync_apply* for a master replica (cf. Algorithm 3, Line 8). In comparison, the original workflow of the computation workers (cf. Algorithm 1) invokes *sync_switch* before the start of the communication, isolating the vertex state used by computation and communication workers and thus guaranteeing lock-free access.

Deferred switching relaxes such isolation, exploiting benign data race to expedite state propagation. For example, in SSSP (cf. Algorithm 4), *sync_switch* involves a copy of the locally-maintained shortest distance. Since this value is monotonically non-increasing, correctness remains intact if functions in *Sync* access a vertex state different from the one that would have been copied by *sync_switch* before the start of a synchronization iteration. It is also beneficial to defer the access of the vertex state during synchronization: the use of an updated state from one host potentially expedites convergence on other hosts. The monotonically non-increasing property, combined with read-only access from the communication workers, justifies benign data race: access to vertex state from both worker groups remains lock-free.

In Hieroglyph, we support both the eager switching design and the deferred switching mode as an optimization.

Local Synchronous/Asynchronous Execution. The CSM abstraction does not define whether locally-sufficient computation should be performed synchronously or asynchronously on each host. Assume PowerGraph GAS for locally-sufficient computation. For an iteration of computation, locally-synchronous execution involves running the gather, apply, and scatter phases each in a dedicated iteration, with a local barrier separating two consecutive phases. Messages produced in the current iteration can be processed only in the subsequent iteration. In locally-asynchronous execution [14, 30, 35], on the other hand, the three phases are applied to a vertex as an integrated function. Messages sent from vertices v_i to v_j are processed in the same iteration, if v_j is processed after v_i.

Locally-synchronous execution achieves lock-free access to vertex and edge data. Locally-asynchronous execution, in contrast, has the advantage of fast state propagation. Hieroglyph supports both execution modes and, for locally-asynchronous execution, supports different consistency models, similar to the vertex/edge/full consistency in distributed GraphLab [22].

Fault-Tolerance. Hieroglyph resorts to a checkpoint-based fault-tolerance mechanism [1, 12, 16]. Checkpoints are created by computation workers independently on each host. The checkpoint interval is specified with respect to synchronization iterations, which are globally consistent. Specifically, state in a checkpoint corresponds to that at the beginning of a metadata synchronization stage. When a fault occurs, all hosts are rolled back to their most recent checkpoints.

Such a design minimizes the graph state stored in a checkpoint: only vertex state and intra-host messages used by *Compute* need to be checkpointed. This is because, at the time of checkpoint creation, there exists no state associated with either *Sync* or *Merge*. The next *Sync* stage is pending, with its initial state yet to be created based on the vertex state from *Compute*. The previous *Merge* has completed, with all its related state incorporated into vertex state. Also recall that no inter-host message exists in locally-sufficient computation. Thus, checkpointing intra-host messages is sufficient for the resumption of locally-sufficient computation.

Optimizations such as computation-checkpointing overlapping apply to Hieroglyph as well. On each host, instead of first checkpointing all related state of its graph partitions and then resuming computation, we can resume the locally-sufficient computation for a vertex, as long as its state and local incoming messages are checkpointed. When such computation produces local outgoing messages, they can be delivered as long as the incoming messages of the receiving vertices are checkpointed. Similarly, inter-host communication and checkpointing can overlap.

Regarding performance overhead, the cost of a fault—measured in the amount of computation lost due to a rollback—is higher for Hieroglyph than general-purpose BSP approaches. This is a direct outcome of the fact that Hieroglyph can progress faster than BSP, thanks to local sufficiency. On the other hand, the overhead of checkpointing in Hieroglyph—in terms of the slowdown of convergence—should be on a par with that of BSP. Assume that (*i*) a fixed checkpointing interval is configured for Hieroglyph and BSP, (*ii*) the cost of generating a checkpoint is the same for both approaches, and (*iii*) an algorithm in Hieroglyph converges $x\%$ faster than BSP. The slowdown attributed to each checkpoint for Hieroglyph is $\frac{100}{100-x}$ of that for BSP. The number of checkpoints taken by Hieroglyph is $\frac{100-x}{100}$ of that of BSP. The two effects negate each other, leading to

[8] Our discussion focuses on threads used by the processing engine layer and does not include those used for background communication.

[9] One iteration of computation over *all* local vertices is performed immediately after an iteration of merging, regardless of chunk configuration, in order to maintain the validity of the termination condition.

Table 4: Graph datasets. Counts in parenthesis are for undirected graphs.

Dataset	$\|V\|$	$\|E\|$
Livejournal	4.8M	69.0M (85.7M)
Wiki	4.2M	101.3M (183.9M)
Twitter	41.7M	1.5B (2.4B)
Road	2.8M	6.8M (6.8M)
Web	118.1M	1.0B (1.7B)

the same slowdown of convergence.

5. EVALUATION

We compare the performance of Hieroglyph with three state-of-the-art graph processing systems and show the superiority of our proposed CSM abstraction.

5.1 Experiment Setup

To evaluate Hieroglyph's performance, we use five realistic datasets (summarized in Table 4). Livejournal [2, 19] describes the friendship relation in the LiveJournal social network. Wiki [3,4] compiles English Wikipedia pages. Twitter [17] captures the "who follows whom" relation in the Twitter social network. Road [11] is the road network of the great lakes area of the United States. Web [3, 4] is a web graph generated by WebBase [8].

We evaluate Hieroglyph with four algorithms: bipartite matching (abbreviated as *Bipart*),[10] weakly connected component (abbreviated as *CC*), PageRank, and SSSP, all common building-block algorithms in graph analytics.

We compare Hieroglyph with PowerGraph [11] [12], PowerLyra [12] [7], and GiraphUC [13] [14]. PowerGraph uses vertex-cut partitioning and features efficient processing of high-degree vertices. PowerLyra extends PowerGraph to support hybrid-cut, enhancing computation efficiency for low-degree vertices. Hieroglyph augments PowerLyra with efficient locally-sufficient computation. Given such relation, regarding both design and implementation, a performance comparison between Hieroglyph and its two predecessors identifies the gain of locally-sufficient computation over vertex-cut. GiraphUC is a vertex-centric graph processing system providing locally-sufficient computation over edge-cut. Despite the discrepancy between GiraphUC and Hieroglyph in terms of implementation details,[14] a comparison between them sheds light on the potential of enabling efficient locally-sufficient computation over vertex-cut partitioning.

All experiments are conducted in a cluster of 16 Amazon EC2 c3.8xlarge instances, each with 32 2.8GHz vCPUs, 60GB memory, and 10 Gbps network connection. All instances run Ubuntu 14.04.2 LTS (Linux 3.13.0-54-generic). PowerGraph, PowerLyra, and Hieroglyph are compiled with gcc 4.8.2. GiraphUC is implemented on Giraph 1.1.0 and run with Hadoop 1.0.4 and jdk 1.7.0_79. Each data point is the mean of at least three runs. For all experiments, grid vertex-cut is used for PowerGraph by default, hybrid-cut with ginger heuristics for PowerLyra and Hieroglyph, and hash-based edge-cut for GiraphUC. PowerGraph and PowerLyra run in the synchronous mode by default.

5.2 Performance

Figure 4 shows that Hieroglyph outperforms all the other three systems for all algorithm-dataset combinations in our evaluation. Hieroglyph's speedup varies from 1.02x to 52.50x, with a median speedup of 3.54x and an average speedup of 6.00x.[15]

Comparing against PowerGraph and PowerLyra. In most settings, the performance improvement of Hieroglyph with respect to both PowerGraph and PowerLyra maximizes on Road and minimizes on Twitter. This is because locally-sufficient computation is most effective with respect to synchronous execution, when local state propagation in synchronous execution is severely hindered by global synchronization. Hieroglyph's effectiveness is thus amplified by Road, which has a large diameter and requires numerous supersteps—each concluded with an iteration of global synchronization—for local state propagation in PowerGraph and PowerLyra.[16]

The diminishing performance gap in Twitter can be understood from the perspective of computation-communication balancing. Given its size, Twitter imposes considerable computation workload on each participating host. On the one hand, it improves the computation-communication interleaving in PowerGraph and PowerLyra, effectively reducing the penalty of inter-host communication. On the other hand, it magnifies the computation overhead of Hieroglyph caused by (*i*) repeated per-vertex computation to process asynchronously-delivered input state and (*ii*) the need for resolving inconsistent local state. Indeed, compared to Livejournal and Wiki, we observe an increase in vertex update rate—an indicator of the computation efficiency—for PowerGraph and PowerLyra in the case of Twitter. The update rate for Hieroglyph, however, drops in the case of Twitter. The opposite trend thus reduces the performance improvement of Hieroglyph.

Results for Bipart show the superiority of Hieroglyph due to its ability to perform locally-sufficient computation across phase boundaries. Given that each of the four phases in Bipart consists of only one superstep and that messages generated in one phase are always processed by the next phase, Hieroglyph's performance would be identical to that of PowerLyra if local sufficiency is confined within phase boundaries. In other words, system-only support for locally-sufficient computation—without the flexibility introduced in the CSM abstraction—would miss the opportunity of performance enhancement in Bipart. With our CSM bipartite matching design, Hieroglyph achieves up to 3.58x speedup over PowerGraph and 3.49x over PowerLyra.

Comparing against GiraphUC. GiraphUC achieves better performance than both PowerGraph and PowerLyra for all four algorithms on LiveJournal, Wiki, and Road. It, however, becomes inferior on Twitter and Web. GiraphUC's performance variation mirrors that of Hieroglyph. When the computation workload increases in the synchronous execution, the relative effectiveness of local sufficiency reduces.

Yet, speedups of Hieroglyph over GiraphUC for Twitter and Web

[10]Treating vertices with an even id as left vertices and the rest as right vertices enables the evaluation of Bipart on all five datasets [26].

[11]We evaluate GraphLab PowerGraph version 2.2 (March 2014).

[12]We use PowerLyra release in April 2014.

[13]We use GiraphUC snapshot in May 2015.

[14]Such discrepancy includes different vertex-centric abstractions, programming languages, and inter-host communication mechanisms.

[15]GiraphUC runs in BSP mode (i.e., reducing to Giraph) in our Bipart measurement. This is because GiraphUC terminates prematurely when executing our Bipart algorithm with locally-sufficient computation. Note, however, that GiraphUC's execution time obtained in BSP mode lower-bounds that with locally-sufficient computation enabled, given the one-superstep-per-phase property of Bipart.

[16]Such pattern conforms to observations in GiraphUC [14].

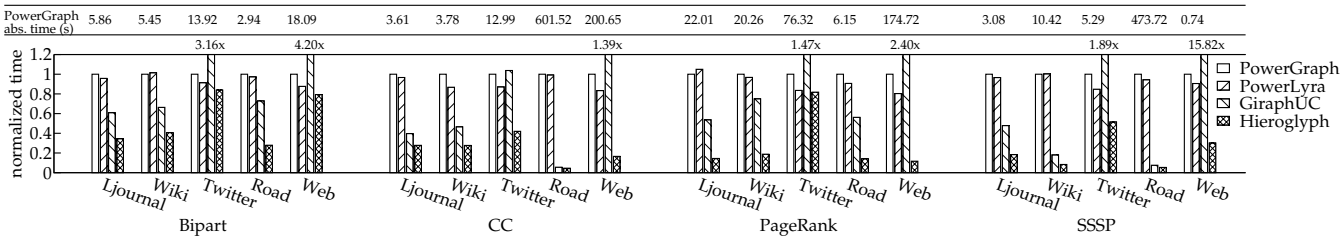

Figure 4: Performance comparison. Execution time is normalized to that of PowerGraph. Absolute average execution time of PowerGraph (in seconds) is marked atop each cluster. Normalized execution time of GiraphUC, when exceeding the plotting range, is also marked.

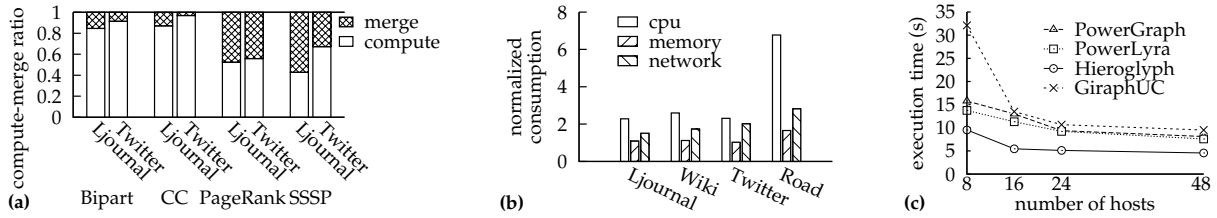

Figure 5: Results on performance breakdown, resource consumption, and scalability. (a) Performance breakdown of Hieroglyph's *Compute* and *Merge*. (b) Resource consumption, normalized to PowerLyra. (c) Scalability comparison, with the number of hosts varying from 8 to 48.

are larger than those for the remaining datasets. Although the reduced effectiveness of locally-sufficient computation affects both Hieroglyph and GiraphUC in Twitter and Web, Hieroglyph, with its vertex-cut support via CSM, gracefully handles the increasing workload with respect to GiraphUC.

Performance Breakdown. Figure 5a shows the performance breakdown of the compute workers, which are responsible for both the compute and the merge stage. For the four algorithms used in our evaluation, the compute stage dominates the workload of the compute workers in most cases. The compute-merge ratio is a function of algorithm and dataset. Figure 5a shows that, (*i*) the compute-merge ratio related to the Twitter dataset is higher than that of Livejournal and (*ii*) the ratios related to Bipart and CC are higher than those of PageRank and SSSP.

In general, a larger portion of execution time in the merge stage indicates a higher cost of inconsistency resolution. This cost is determined by two factors: the frequency of the activation of merge stages and the cost of each activation with respect to the cost of compute stages. In the case of SSSP, for example, given that the costs of each activation of the merge stage and the compute stage are comparable, the large portion of execution time in the merge stage indicates frequent activation of the merging logic. Note, however, that both compute and merge stages contribute to the final graph state convergence. A low compute-merge ratio does not entail an insufficient CSM algorithm design.

Resource Consumption. Figure 5b shows the resource consumption of Hieroglyph with respect to PowerLyra when executing SSSP. CPU consumption is measured by the number of vertex state updates. Hieroglyph conducts a substantially larger amount of vertex updates, due to the activation of the update function on all replicas of each vertex, instead of only the master replica. Such overhead is also due to the use of potentially inconsistent local state for update in Hieroglyph. Inconsistency resolution incurs a 51%–182% overhead regarding network traffic. Since the communication workers progress independently, however, the negative impact of such an overhead on the overall performance is minimized. Hieroglyph's memory overhead varies from 3% to 65%, thanks to the maintenance of additional states for *Sync* and *Merge*.

Table 5: Hieroglyph's speedup over synchronous and asynchronous PowerGraph and PowerLyra

CC	PowerGraph	PowerLyra
synchronous	2.38x	2.07x
asynchronous	8.27x	6.15x
SSSP	**PowerGraph**	**PowerLyra**
synchronous	1.94x	1.65x
asynchronous	17.42x	18.58x

Scalability. Figure 5c shows the scalability of the four systems when executing CC on Twitter. The execution time of all systems reduces with the increasing number of hosts. Yet, all systems demonstrate sublinear speedup with the increasing number of hosts, due to the intrinsic inter-host dependency of the workload. In terms of execution time, Hieroglyph outperforms the other systems in all our settings. Its speedup varies between 1.66x–3.37x, 2.38x–2.47x, 1.83x–2.08x, and 1.78x–2.08x, when the number of hosts are 8, 16, 24, and 48, respectively.

Asynchronous Execution. When running in the asynchronous mode, the performance of PowerGraph and PowerLyra degrades significantly for CC, PageRank, and SSSP.[17] Figure 6a compares the performance of PowerGraph, PowerLyra, and Hieroglyph when executing CC and SSSP on Twitter (summarized in Table 5). We choose Twitter for evaluating the asynchronous mode of PowerGraph and PowerLyra, because it produces the minimum speedup for Hieroglyph and thus provides a conservative view of the performance improvement.

[17]Bipart requires synchronous mode on PowerGraph and PowerLyra.

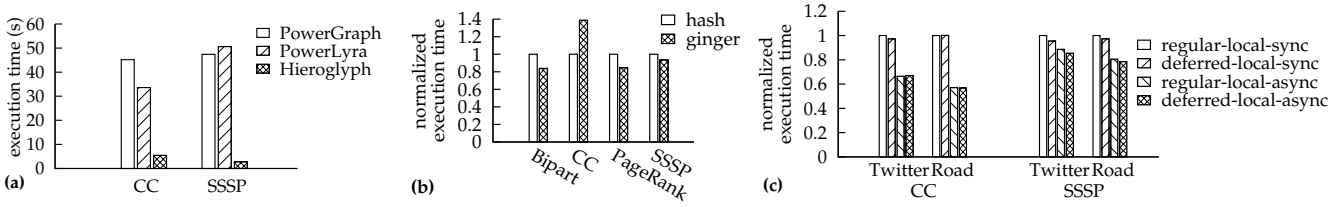

Figure 6: Results on asynchronous execution, effects of graph partitioning, and Hieroglyph's optimization using deferred switching and locally-synchronous and asynchronous modes. (a) Performance of PowerGraph and PowerLyra in asynchronous mode. (b) Effect of graph partitioning. (c) Effect of regular vs. deferred switching and locally-synchronous vs. asynchronous execution in Hieroglyph .

For CC, the speedup of Hieroglyph with respect to PowerGraph more than triples when comparing the latter's synchronous execution to asynchronous execution. As for PowerLyra, the speedup of Hieroglyph triples, as well. The increase is even more significant in the case of SSSP. Hieroglyph's speedup boosts from 1.94x for synchronous PowerGraph to 17.42x for asynchronous PowerGraph, enlarging the performance gap by 8x. For PowerLyra, the gap enlarges by over an order-of-magnitude.

Graph Partitioning. Figure 6b shows the performance of Hieroglyph using hash and ginger-heuristic graph partitioning [7], measured with the Twitter dataset. Overall, Hieroglyph achieves high performance in both hash and ginger partitioning. Given its ability to perform locally-sufficient computation, Hieroglyph can mitigate the effect of unbalanced workload caused by graph partitioning. Yet, in general, Hieroglyph still benefits from more balanced ginger partitioning.

Deferred Switching. The effect of deferring the switching of synchronization state from the beginning of a synchronization iteration to the time when such state is accessed is negligible for CC and SSSP, yielding a maximum of 5% reduction in execution time (cf. Figure 6c). Such ineffectiveness may be partly attributed to the high priority assigned to the communication workers in the current implementation of Hieroglyph. While the computation workers frequently yield to the communication workers (e.g., by reducing the vertex chunk size), the latter proceed until all available data have been exchanged. The rationale behind this default mode of Hieroglyph is that, when local state is updated, it is advantageous to propagate the update to all replicas. In other words, it is desirable to minimize the time window during which the vertex state remains inconsistent. Locally-sufficient computation proceeds opportunistically, aiming at making progress to hide communication cost yet minimizing additional communication delay (in the form of reduced responsiveness of the communication workers due to parallel locally-sufficient computation). The probability that the local vertex state is repeatedly updated before synchronization is thus minimized, so is the effect of deferred switching.

There are, nevertheless, cases where it is beneficial to assign high priority to local state propagation [35].[18] In those cases, we expect deferred switching to significantly shorten the execution time.

Local Asynchrony. Figure 6c also compares the performance of locally-synchronous execution with that of locally-asynchronous execution. For all cases in Figure 6c, local asynchrony leads to superior performance, with the speedup ranging from 1.13x to 1.76x. For both CC and SSSP, the gain of local asynchrony is larger for Road than for Twitter. This is because the effect of fast state propagation in locally-asynchronous mode is amplified by the large diameter of Road.

It is also worth noting that, for CC and SSSP, vertex-consistency is sufficient for correctness [22]. Since during local-sufficient computation, vertex state can be updated only in the *apply* function, there is no write-write data race. In addition, read-write data race is benign in both CC and SSSP.[19] Consequently, no lock is required for accessing vertex state. Regarding operations on the message queue, message enqueuing requires lock protection in both locally-synchronous and asynchronous modes. The only additional locking overhead induced by locally-asynchronous execution is thus for message dequeuing. This slight overhead is outweighed by the benefit of fast state propagation, leading to the significant improvement of locally-asynchronous execution.

The performance improvement of local asynchrony is encouraging. Yet, locally-synchronous execution has its own merit. For example, it efficiently supports the bipartite matching algorithm, in which active vertices of the current phase send messages to be processed by the subsequent phase. Had Hieroglyph only supported locally-asynchronous mode, it would require the implementation of phase-related message tagging [14], complicating multi-phase algorithm design.

Table 6: Algorithm Complexity in Lines of Code

Algorithm	GAS	CSM
Bipart	123	206
CC	46	88
PageRank	48	99
SSSP	44	88

Complexity. Table 6 compares the complexity of CSM algorithms with their GAS counterparts, using lines of code as the metric. Using Hieroglyph's implementation of the CSM abstraction, the four algorithms studied in the evaluation require 67%–106% more lines of code to be expressed in CSM. Note that, the complexity of algorithm design in CSM is also determined by the inconsistency resolution logic in *Sync* and *Merge*. For CC, PageRank, and SSSP, their corresponding inconsistency resolution logic resembles the logic used in their locally-sufficient computation, the latter an extension

[18] In graph-centric approaches [30, 35], the same problem bares the form of whether to perform per-partition computation iteratively (e.g., until the partition converges) or to frequently propagate updated external vertex state to adjacent partitions.

[19] For example, assume that a vertex v_i will not send a message to another vertex v_j, if v_i obtains (i.e., reads) the most recent update (i.e., write) of v_j. Then, if v_i sends a message to v_j due to the access of a stale state of v_j but the message arrives after v_j's update, it will be discarded due to program logic, for both CC and SSSP. Such race thus does not affect correctness.

of the GAS implementation. As a result, these three algorithms are relatively easy to be implemented. Bipart is more difficult, in contrast, because of the dissimilarity between *Compute* and the inconsistency-fixing logic in *Sync* and *Merge*.

6. RELATED WORK

We have discussed Pregel [23], PowerGraph [12], PowerLyra [7], and GiraphUC [14]. Below we summarize other graph processing systems.

Execution Modes. Many graph processing systems, such as Giraph [1], Mizan [16], GPS [27], Pregel+ [37], GraM [33], Quegel [38], and Version Traveler [15], follow the bulk synchronous parallel model. Giraph [1] is an open-source implementation of Pregel. Mizan [16] features dynamic workload balancing. GPS [27] supports master computation—computation performed by a master host and serialized to BSP supersteps on all hosts—and introduces dynamic graph repartitioning and large adjacency list partitioning for reducing communication overhead. Pregel+ [37] analyzes the benefit of vertex state mirroring [22] and extends the Pregel abstraction with a request-respond paradigm, enhancing the flexibility in state propagation. GraM [33] achieves overlapping of computation and communication at the architectural level, via a multi-core-aware RDMA-based communication stack. Quegel [38] extends BSP to support superstep-sharing execution, effectively amortizing the cumulative synchronization cost across the parallel execution of multiple queries. Version Traveler [15] enables fast version switching in multi-version graph processing.

Besides BSP-style systems, PowerGraph [12], Trinity [28], and GRACE [32] support both synchronous and asynchronous modes. PowerSwitch [34] employs Hsync, a hybrid mode featuring adaptive switching between synchronous and asynchronous modes for better performance. Several parameter server frameworks, such as LazyTable [10] and the work of Li *et al.* [20], explore the stale synchronous parallel (SSP) model [9]—a relaxed synchronous model achieving high communication efficiency and bounded staleness.

None of the above systems supports local sufficiency over vertex-cut with a vertex-centric abstraction. Hieroglyph enhances state of the art by closing this critical gap.

Graph-Centric Programming. Graph-centric programming (e.g., Giraph++ [30] and Blogel [36]) exposes graph partitions to users, enabling more efficient algorithm design. The use of per-partition local input state during computation resembles local sufficiency. The two differ, nevertheless, in the following aspects. Graph-centric programming, by definition, diverges from vertex-centric programming. It is synchronous in that computation proceeds in supersteps and computation and communication—over all vertices of a partition and/or the partition itself—alternate. It reduces the communication overhead via partition-oriented algorithm redesign. Local sufficiency is, in contrast, vertex-centric and intrinsically asynchronous due to the independence between computation and communication. It hides the communication overhead behind computation. Despite the substantial performance gain witnessed by prior work on graph-centric programming [30,35,36], vertex-centric programming remains dominant due to simplicity.

Multi-Core/Out-of-Core Processing. Approaches towards efficient multi-core [29] and out-of-core processing [18,24,25,39,40], are orthogonal to Hieroglyph, the latter targeting a distributed in-memory scenario.

Dataflow Operators. Pregelix [5] and GraphX [13] both map a vertex-centric abstraction onto distributed dataflow operators, enabling graph processing over general purpose dataflow engines. Mapping CSM onto dataflow operators would enable locally-sufficient computation on graph processing systems built atop dataflow engines, thus improving the latter's performance in the graph processing stage.

7. CONCLUSIONS

In this paper, we introduced Compute-Sync-Merge, a vertex-centric abstraction supporting efficient locally-sufficient computation. CSM enforces local sufficiency at the abstraction level and supports vertex-cut by activating vertex-centric computation on all vertex replicas. We demonstrated the expressiveness of CSM by implementing several fundamental algorithms. Hieroglyph—our CSM-compliant prototype system—outperforms state of the art by up to 53x.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd, Abhishek Chandra, for their feedback. The work reported in this paper was supported in part by Intel Corporation.

9. REFERENCES

[1] Apache Giraph. http://giraph.apache.org.

[2] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan. Group formation in large social networks: Membership, growth, and evolution. In *KDD '06*, pages 44–54, 2006.

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *WWW'11*, 2011.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *WWW'04*, pages 595–601, 2004.

[5] Y. Bu, V. Borkar, J. Jia, M. J. Carey, and T. Condie. Pregelix: Big(ger) graph analytics on a dataflow engine. *PVLDB*, 8(2):161–172, 2014.

[6] R. Chen, X. Ding, P. Wang, H. Chen, B. Zang, and H. Guan. Computation and communication efficient graph processing with distributed immutable view. In *HPDC '14*, pages 215–226, 2014.

[7] R. Chen, J. Shi, Y. Chen, and H. Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. In *EuroSys '15*, pages 1:1–1:15, 2015.

[8] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley. Stanford webbase components and applications. Technical Report 2004-34, Stanford InfoLab, 2004.

[9] J. Cipar, Q. Ho, J. K. Kim, S. Lee, G. R. Ganger, G. Gibson, K. Keeton, and E. Xing. Solving the straggler problem with bounded staleness. In *HotOS'13*, pages 22–22, 2013.

[10] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing. Exploiting bounded staleness to speed up big data analytics. In *USENIX ATC'14*, pages 37–48, 2014.

[11] DIMACS. 9th dimacs implementation challenge - shortest paths. http://www.dis.uniroma1.it/challenge9/download.shtml.

[12] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. Powergraph: Distributed graph-parallel computation on natural graphs. In *OSDI'12*, pages 17–30, 2012.

[13] J. E. Gonzalez, R. S. Xin, A. Dave, D. Crankshaw, M. J. Franklin, and I. Stoica. Graphx: Graph processing in a distributed dataflow framework. In *OSDI '14*, pages 599–613, Oct. 2014.

[14] M. Han and K. Daudjee. Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. *PVLDB*, 8(9):950–961, May 2015.

[15] X. Ju, D. Williams, H. Jamjoom, and K. G. Shin. Version traveler: Fast and memory-efficient version switching in graph processing systems. In *USENIX ATC'16*, pages 523–536, 2016.

[16] Z. Khayyat, K. Awara, A. Alonazi, H. Jamjoom, D. Williams, and P. Kalnis. Mizan: A system for dynamic load balancing in large-scale graph processing. In *EuroSys '13*, pages 169–182, 2013.

[17] H. Kwak, C. Lee, H. Park, and S. Moon. What is twitter, a social network or a news media? In *WWW '10*, pages 591–600, 2010.

[18] A. Kyrola, G. Blelloch, and C. Guestrin. Graphchi: Large-scale graph computation on just a pc. In *OSDI'12*, pages 31–46, 2012.

[19] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney. Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters. *Internet Mathematics*, 6(1):29–123, 2009.

[20] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI'14*, pages 583–598, 2014.

[21] Y. Low. *GraphLab: A Distributed Abstraction for Large Scale Machine Learning*. PhD thesis, Carnegie Mellon University, 2013.

[22] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. Distributed graphlab: A framework for machine learning and data mining in the cloud. *PVLDB*, 5(8):716–727, Apr. 2012.

[23] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A system for large-scale graph processing. In *SIGMOD '10*, pages 135–146, 2010.

[24] A. Roy, L. Bindschaedler, J. Malicevic, and W. Zwaenepoel. Chaos: Scale-out graph processing from secondary storage. In *SOSP '15*, pages 410–424, 2015.

[25] A. Roy, I. Mihailovic, and W. Zwaenepoel. X-stream: Edge-centric graph processing using streaming partitions. In *SOSP '13*, pages 472–488, 2013.

[26] S. Salihoglu, J. Shin, V. Khanna, B. Q. Truong, and J. Widom. Graft: A debugging tool for apache giraph. In *SIGMOD '15*, pages 1403–1408, 2015.

[27] S. Salihoglu and J. Widom. Gps: A graph processing system. In *SSDBM*, pages 22:1–22:12, 2013.

[28] B. Shao, H. Wang, and Y. Li. Trinity: A Distributed Graph Engine on a Memory Cloud. In *SIGMOD'13*.

[29] J. Shun and G. E. Blelloch. Ligra: A lightweight graph processing framework for shared memory. In *PPoPP '13*, pages 135–146, 2013.

[30] Y. Tian, A. Balmin, S. A. Corsten, S. Tatikonda, and J. McPherson. From "think like a vertex" to "think like a graph". *PVLDB*, 7(3), 2013.

[31] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.

[32] G. Wang, W. Xie, A. J. Demers, and J. Gehrke. Asynchronous large-scale graph processing made easy. In *CIDR '13*, 2013.

[33] M. Wu, F. Yang, J. Xue, W. Xiao, Y. Miao, L. Wei, H. Lin, Y. Dai, and L. Zhou. Gram: Scaling graph computation to the trillions. In *SoCC '15*, pages 408–421, 2015.

[34] C. Xie, R. Chen, H. Guan, B. Zang, and H. Chen. Sync or async: Time to fuse for distributed graph-parallel computation. In *PPoPP 2015*, pages 194–204, 2015.

[35] W. Xie, G. Wang, D. Bindel, A. Demers, and J. Gehrke. Fast iterative graph computation with block updates. *PVLDB*, 6(14):2014–2025, Sept. 2013.

[36] D. Yan, J. Cheng, Y. Lu, and W. Ng. Blogel: A block-centric framework for distributed computation on real-world graphs. *PVLDB*, 7(14):1981–1992, Oct. 2014.

[37] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective techniques for message reduction and load balancing in distributed graph computation. In *WWW '15*, 2015.

[38] D. Yan, J. Cheng, M. T. Özsu, F. Yang, Y. Lu, J. C. S. Lui, Q. Zhang, and W. Ng. A general-purpose query-centric framework for querying big graphs. *PVLDB*, 9(7):564–575, Mar. 2016.

[39] P. Yuan, W. Zhang, C. Xie, H. Jin, L. Liu, and K. Lee. Fast iterative graph computation: A path centric approach. In *SC '14*, pages 401–412, 2014.

[40] D. Zheng, D. Mhembere, R. Burns, J. Vogelstein, C. E. Priebe, and A. S. Szalay. Flashgraph: Processing billion-node graphs on an array of commodity ssds. In *FAST'15*, pages 45–58, 2015.

APPENDIX

Correctness of Bipartite Matching in Hieroglyph. The CSM implementation of bipartite matching is based on the original Pregel implementation—a randomized maximal matching algorithm. Its correctness is determined by the properties of the final graph state.

DEFINITION 1. *A distributed randomized maximal bipartite matching algorithm is correct if, upon graph convergence, (i) all replicas of a vertex are in the same state; (ii) if a vertex is matched, then it matches with a vertex on the other side of the bipartite graph; (iii) if a vertex v_i matches with another vertex v_j, then v_j matches with v_i; and (iv) if a vertex is unmatched, then all its neighbors on the other side of the graph, if any, are matched.*

Regarding the four properties of the final graph state, the first addresses general consistency of distributed graph state. The second property is the requirement of bipartite matching. The third property is the consistency requirement of a matching. The fourth property is the requirement of a maximal matching.

THEOREM 1. *The CSM implementation of bipartite matching is correct.*

We prove Theorem 1 by proving the following lemmas.

LEMMA 1. *Upon graph convergence, all replicas of a vertex are in the same state.*

PROOF. The final state of all replicas of a vertex v_i is the same as the state decided by the arbitration logic in the last synchronization iteration k related to v_i. After that iteration of synchronization, there exists no local update to v_i at any host. This is because, any further local update to a replica of v_i after k, due to either computation or revocation, leads to another synchronization iteration $k+1$ related to v_i, contradicting with the assumption that k is the last synchronization for v_i.

The consistent final state of v_i is thus an outcome of the arbitration logic, which selects one and only one value from all replicas of v_i. □

LEMMA 2. *Upon graph convergence, if a vertex is matched, then it matches with a vertex on the other side of the bipartite graph.*

PROOF. All matching decisions are results of local matching decisions, direct (on the same host where the local matching takes place) or indirect (via propagation). Local matching decisions are made according to the original proven-correct bipartite matching algorithm used in Pregel. Left vertices thus only match with right vertices, and vice versa. □

LEMMA 3. *Upon graph convergence, if a vertex v_i matches with another vertex v_j, then v_j matches with v_i.*

PROOF. Suppose, in the final graph state, v_i matches with v_j but v_j does not match with v_i.

For v_i, the matching between v_i and v_j stems from a local matching decision made by a host h maintaining the edge connecting v_i and v_j. The last state update of v_i on h must be because of the local matching decision. Note the time when the last update of v_i on h takes place as t.

At t, v_j's local state at h is "matched with v_i," due to the local matching decision. Since v_j's final state is "not matched with v_i," the local state of v_j on h must be updated at t', $t < t'$. At t', v_j sends a local revocation message to v_i, according to the two-phase merging logic. Such revocation message must be relevant to v_i, because v_i's state remains "matched with v_j" after t. This relevant revocation message will then cause v_i's local state to change, contradicting with the assumption that v_i's state is final after t. □

LEMMA 4. *Upon graph convergence, if a vertex is unmatched, then all its neighbors on the other side of the graph, if any, are matched.*

PROOF. Support v_i and v_j are neighbors on opposite sides of the graph and both unmatched in the final graph state.

There are two cases where v_i and v_j can be both unmatched in the final graph. First, v_i and v_j remain unmatched during the course of the computation. Second, at least one of them becomes matched during the computation but then becomes unmatched due to CSM's revocation logic.

In the first case, there must be a host h maintaining the edge connecting v_i and v_j. As a result, the two vertices cannot remain unmatched on h, according to the proven-correct local matching algorithm.

In the second case, without loss of generality, assume that v_i enters the final unmatched state no earlier than v_j. When v_i enters the final unmatched state, it activates itself and/or v_j on h, according to the two-phase merging logic. After the activation, the two remain unmatched on h—an impossible condition according to the proven-correct local matching algorithm. □

According to Lemmas 1–4, we have Theorem 1 by Definition 1.