An Algorithmic Approach to Communication Reduction in Parallel Graph Algorithms

Harshvardhan[†], Adam Fidel, Nancy M. Amato, Lawrence Rauchwerger
Parasol Laboratory
Dept. of Computer Science and Engineering
Texas A&M University
{ananvay, fidel, amato, rwerger}@cse.tamu.edu

***Abstract*—Graph algorithms on distributed-memory systems typically perform heavy communication, often limiting their scalability and performance. This work presents an approach to transparently (without programmer intervention) allow fine-grained graph algorithms to utilize algorithmic communication reduction optimizations. In many graph algorithms, the same information is communicated by a vertex to its neighbors, which we coin algorithmic redundancy. Our approach exploits algorithmic redundancy to reduce communication between vertices located on different processing elements. We employ algorithm-aware coarsening of messages sent during vertex visitation, reducing both the number of messages and the absolute amount of communication in the system. To achieve this, the system structure is represented by a hierarchical graph, facilitating communication optimizations that can take into consideration the machine's memory hierarchy. We also present an optimization for small-world scale-free graphs wherein *hub vertices* (i.e., vertices of very large degree) are represented in a similar hierarchical manner, which is exploited to increase parallelism and reduce communication. Finally, we present a framework that transparently allows fine-grained graph algorithms to utilize our hierarchical approach without programmer intervention, while improving scalability and performance. Experimental results of our proposed approach on $131,000+$ cores show improvements of up to a factor of 8 times over the non-hierarchical version for various graph mining and graph analytics algorithms.**

***Keywords*—parallel graph processing; graph analytics; big data;**

I. Introduction

Graph algorithms are used in many real-world problems, from mining social networks and big-data analytics to scientific computing where meshes are used to model physical domains. With the advent of big data, there has been a significant interest in improving the scalability of graph algorithms. Moreover, certain graphs of interest – such as web-graphs and social networks – exhibit small-world scale-free characteristics with a power-law degree distribution and small diameters. While regular and irregular meshes can be partitioned to decrease the edge-cut between partitions, partitioning scale-free graphs is difficult due to the presence of *hub vertices* with very high out-degrees. As communication in many algorithms is directly proportional to the number of cut-edges, these types of graphs experience heavy communication, severely limiting scalability.

A key pattern that arises in many parallel graph algorithms is the need for a vertex to propagate the same information to all vertices in its neighborhood. For example, consider the traditional fine-grained expression of the breadth-first search algorithm in Figure 1. First, the *vertex-operator* (Figure 1(a)) checks to see if a vertex is active (grey), and if so, it propagates its distance from the source to its neighbors using the *neighbor-operator* (Figure 1(b)). The neighbor-operator then updates the distance of the neighbor if needed, and marks the neighbor as active (grey) in the next iteration. A crucial observation is that the vertex-operator visits all of its neighbors with semantically identical information, which may lead to redundant communication in distributed-memory architectures. This communication pattern is present in a large class of important algorithms such as breadth-first search, betweenness centrality, connected components, community detection, PageRank, k-core decomposition, triangle counting, etc. These algorithms are used in graph mining and big-data applications where performance is critical.

```
bool bfs_vertex_op(vertex v)
  if (v.color == GREY)          // Active if GREY
    v.color = BLACK;
    VisitAllNeighbors(bfs_neighbor_op(_1, v.dist+1), v);
    return true;                // vertex was Active
  else  return false;           // vertex was Inactive
```

(a) vertex-operator

```
bool bfs_neighbor_op(vertex u, int new_distance)
  if (u.dist > new_distance)
    u.dist = new_distance;  // update distance
    u.color = GREY;         // mark to be processed
    return true;            // vertex was updated
  else  return false;
```

(b) neighbor-operator

Fig. 1. The vertex- and neighbor-operators for breadth-first search.

The cost of memory accesses on large-scale distributed-memory systems is highly non-uniform. The communication patterns exhibited in graph algorithms executed on these systems are well-known to limit scalability. For this situation, optimizations to alleviate communication pressure can occur in several forms:

- Messages destined to the same processing element can be aggregated and combined into a single message.
- Creation of redundant messages that are a result of algorithmic redundancy (such as in breadth-first search) can be eliminated completely, and a single copy can instead be sent between processing elements.
- Bottlenecks caused by the presence of hub vertices can be mitigated by reducing incoming communication to hubs locally, and a single contribution can then be forwarded off-processor.

[†]Now at Google Inc., Mountain View, CA, USA.

However, as algorithms are best expressed in a fine-grained manner that makes them oblivious to the graph structure, and as graphs themselves are represented as flat data-structures, only the first of these three optimizations is typically applied, as the knowledge of machine hierarchy is unavailable.

Our approach is to allow algorithms to be expressed in the natural vertex-centric manner while transparently applying communication optimizations without programmer intervention. The mechanism by which this is achieved is through the construction of a hierarchical representation of the input graph that is aligned with the machine hierarchy to identify local and non-local elements. By using this hierarchical representation and algorithm-level knowledge, we are then able to identify and transparently apply these communication optimizations for fine-grained algorithms.

Our contributions include:

- A software framework to transparently (i.e., without user intervention) allow fine-grained graph algorithms to execute in a manner that is machine-hierarchy aware, enabling important communication optimizations to be applied.
- A novel algorithmic approach to reduce communication, both in number of messages sent and *total amount of data* sent by exploiting algorithmic redundancy observed in parallel graph algorithms.
- Experimental evaluation on two large-scale systems at $12,000+$ and $130,000+$ cores, with several important graph mining and graph analytics algorithms, such as breadth-first search, connected components, k-core and PageRank showing improvements of $2.5\times$ to $8\times$ over traditional approaches.

II. Our Approach

Our goal is to improve scalability and performance of graph algorithms by reducing the number and total volume of messages sent during execution. We first discuss locality-based communication optimizations for graphs in Section II-A and then follow with an additional optimization suited specifically for reducing bottlenecks due to hubs in Section II-B.

A. Locality-Based Communication Optimization

Communication in graph algorithms occurs between vertices through edges, which represent a source-target pair (s, t). For edges with the source s and with targets $t_0, t_1, \ldots$ stored on the same destination location, it is possible to aggregate messages for all such equivalent edges. Further, if all messages represent the same information, a single message can be sent to the destination location and then applied to $t_0, t_1, \ldots$. This concept can be applied recursively by considering a grouping of vertices and identifying same destination location pairs amongst these groupings. The same communication reduction techniques apply to all levels of this recursive process.

To enable communication reduction, we need a model of the system that captures the locality of the graph. To achieve this, we overlay the input graph on top of the machine hierarchy, thereby creating a hierarchically coarsened graph. This is described in Section II-A1. Once the graph is coarsened, we use a translation layer, called the hierarchical paradigm (Algorithm 2), to execute the fine-grained algorithm on the coarsened hierarchy. This is described in Section II-A2. Combining the fine-grained specification with the hierarchy results in a coarsening of the algorithm itself, and allows for the use of algorithm-driven communication optimizations.

1) Hierarchy Construction: The transformation of the flat input graph to a hierarchical graph based on the machine-locality information is a mutating process that groups the graph's vertices into partitions based on the locality of each vertex provided by the machine hierarchy. This replaces multiple vertices in a partition, along with intra-partition edges, with a single super-vertex representing the underlying sub-graph. All edges between two partitions are replaced with a super-edge (composite edge) representing the coarsened communication between two sub-graphs. The super- vertices and super-edges form a super-graph. This process is shown in Figure 2 and detailed in this section.

A hierarchical graph consists of two or more levels of graphs, where the graph at level i is a super-graph of the graph at level $i-1$. The lowest level is the base-level. For building our hierarchy, we first partition the input graph into sub-graphs, where each sub-graph has a similar number of vertices, and assign each sub-graph to a processor. This partitioning can be computed with an external partitioner to reduce the edge-cut of the graph, which may improve the overall performance, while the hierarchical approach will take care of the remaining cross-edges. This forms the base-level of our hierarchical representation (G_0). Next, we create a hierarchy of M levels on this, matching the machine hierarchy. For graph G_i at every level ($0 \leq i \leq M$), we create a super-graph G_{i+1}, such that each vertex in G_{i+1} represents a sub-graph partition of G_i. Super-edges are added between two super-vertices of G_{i+1} if there exist inter-partition edges of their corresponding sub-graphs in G_i. The inter-partition edges of lower-level graph G_i are then removed and their information is stored on the corresponding target's super-edge. This is done to preserve the locality of edges, as all edges that point to the same target are stored at that target vertex's processor.

Our hierarchical graph consists of the base-level graph, along with one or more levels of super-graphs of the base-level graph, with super-vertices representing each vertex-partition and super-edges representing inter-partition edges (communication). The hierarchical representations obtained thus naturally expresses the machine topology. We note that creating the hierarchy is a one-time event that happens immediately following graph construction. Once the hierarchy is created, multiple algorithms can take advantage of it.

2) Using the Hierarchy: The hierarchical graph paradigm is presented in Algorithm 2. In order to execute algorithms, the paradigm proceeds in *supersteps* similar to the bulk-synchronous parallel (BSP) model [22]. However, the supersteps are executed in a manner that is aware of the machine hierarchy. Within each hierarchical superstep, the paradigm processes the active vertices in each level of the graph hierarchy iteratively. For each active vertex in the base-level graph, the results from processing it are sent to its neighbors in the same partition (lines 4-8). Any cross-partition edges for that level of hierarchy are ignored, as they will be processed by the upper-level of the hierarchy. The results from processing

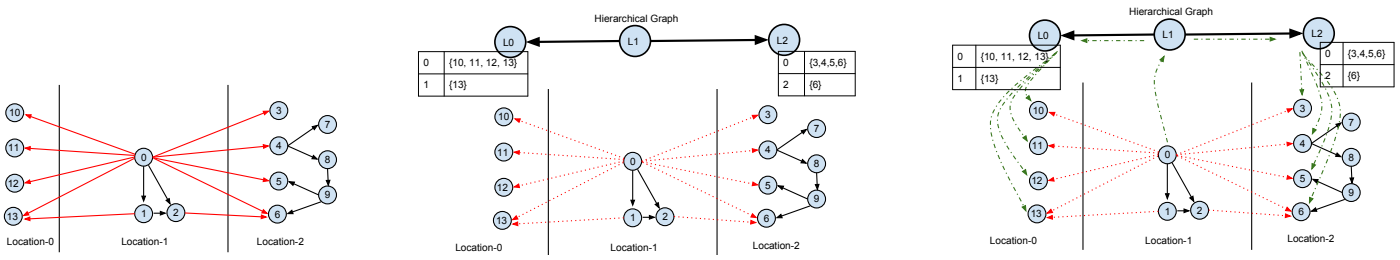


Fig. 2. Creating a hierarchy: (a) A flat graph, with remote, inter-partition edges (red lines) and (b) its hierarchical representation with metadata. Using the hierarchy: (c) Remote communication during algorithm execution (shown in green). Hierarchical graph replaces inter-partition edges in original graph (red dashed-lines represent deleted edges) with a single super-edge between each partition (solid black lines on top-level graph). Generalizable to multiple levels of machine hierarchy.

Algorithm 1 Hierarchical Graph Creation

Input: Graph G_i, int N

1: **if** $i = N$ **then**
2: return
3: **end if**
4: Graph G_{i+1} = Partition(G_i, MachineHierarchy)
5: **for all** $(u, v) \in Edges(G_i)$ **do**
6: **if** $Super_{i+1}(u) \neq Super_{i+1}(v)$ **then**
7: $E_i := E_i \backslash (u, v)$
8: $E_{i+1} := E_{i+1} \bigcup (Super_{i+1}(u), Super_{i+1}(v))$
9: **end if**
10: **end for**
11: Hierarchy.add(G_{i+1})
12: Construct(G_{i+1}, N)

Algorithm 2 Hierarchical Graph Paradigm

Input: Graph G_0, Locality-Hierarchy H, Hubs H_{hubs}

1: **while** active vertices $\neq \emptyset$ **do**
2: **for all** active vertices $v_i \in G_0$ **par do**
3: $W_i \leftarrow$ process(v_i)
4: **for all** neighbors $u_i \in G_0$.adjacents(v_i) **do**
5: **if** H.parent(u_i) = H.parent(v_i) **then**
6: visit-neighbor(u_i, W_i)
7: **end if**
8: **end for**
9: **for all** neighbors $p_i \in H$.adjacents(H.parent(v_i)) **do**
10: async(visit-neighbor(p_i, W_i))
11: // apply visit-neighbor recursively on children of p_i who are neighbors of v_i
12: **end for**
13: **for all** hubs $h_i \in H_{hubs}$.adjacents(v_i) **do**
14: visit-neighbor(h_i, W_i)
15: **end for**
16: **end for**
17: **end while**

the active vertex are then forwarded to the super-vertex of the current partition (lines 9-12), which then transmits them via super-edges to the target super-vertices. If the *process* function in line 3 sends the same update to multiple neighbors (by calling VisitAllNeighbors, for example, as in the case of BFS, PageRank, connected components, k-core, betweenness centrality, etc.), the paradigm only creates a single copy of the update to send via the super-edges. We also provide a specialization for high-degree vertices (described later in Section II-B) that uses a similar communication reduction mechanism (lines 13-15). Finally, the updates are then recursively pushed to the respective target vertices in the lower-level graphs at the target partition, whose edges were replaced by the super-edge.

The graph algorithms themselves do not change and are hierarchy-oblivious, allowing us to reuse fine-grained vertex-centric algorithms on coarsened graphs. This decouples users from the details of machine and locality exploitation.

B. Distributed Hubs Optimization

Hub vertices, or hubs, are vertices with a high in- or out- degree. Many small-world scale-free graphs, such as web-graphs and social-networks, exhibit a power-law degree distribution, with most vertices connected to a few other vertices, but very few ($< 1\%$) vertices connected to an extremely large number of vertices. While our locality-based hierarchy reduces outgoing communication caused by hub vertices, communication can be further reduced by specialized optimizations for high in-degree vertices.

We introduce a new approach to reduce all messages to a hub from the same processor to a single value which can then be applied to the hub. We assume that the applied operator is both associative and commutative. When a vertex tries to update (send a message to) a hub vertex, the operator is applied to a local representative for that hub. At the end of each superstep of the algorithm, the results of the local updates are flushed and applied to the original hubs. This in effect distributes the work for hubs across the system.

Construction of the distributed hubs occurs by first identifying the hub vertices using a simple scan through the graph, and then creating a super-graph where each super-vertex contains metadata about the hub vertices, and all edges to the hub are replaced by the super-edge. This metadata can only be written to and not read from, which does not require the value to be kept coherent, and works with our asynchronous update model (Section IV-B).

We handle the identification of hubs and creation of the hierarchy automatically using the degree (number of edges) of each vertex. The user may choose to provide a cutoff for this size, beyond which vertices will be treated as hubs, or they may choose to use the top k% vertices as hubs. In either case, the framework identifies the hubs so algorithm-developers do not need to account for this.

This approach is similar to recent work presented in [20], which replicates the hub vertices on other processors (called hub-representatives), allowing local vertices to read from and write to the hub vertices. However, their approach replicates data for each hub vertex, which needs to be kept synchronized, leading to extra communication and affecting scalability. Their approach also requires algorithms to be aware of the existence of these hub representatives and need to be modified to specify the synchronizing and reduction behaviors of the representatives. In contrast, our approach is framework-level, and keeps the algorithm itself agnostic to hubs. Further, our approach does not replicate vertex data.

III. Modeling

In this section, we describe how we exploit both the redundant nature of many parallel graph algorithms and the power-law characteristics of many input graphs, resulting in a reduction of total communication in the system.

A. Communication Reduction

The hierarchical approach can reduce communication in the system by reducing the number of bytes required to update neighboring vertices. Without loss of generality, we assume a non-multi-edged graph where two vertices are connected by at most one edge, and that the graph algorithm visits each vertex (and consequently every edge, due to the non-multi-edged property). We also assume a BSP/level-synchronous model [22], [17] where an active vertex performs some computation and updates its neighboring vertices with the result of this computation. The BSP model gives the most conservative estimate for this analysis. Any algorithm that communicates more, such as in an asynchronous model with redundant work, will observe a greater communication reduction using our approach. Let us assume the size (in bytes) of this result is r $bytes$, and that it is uniform for all vertices. In the traditional BSP case, used in existing graph libraries such as Pregel [17], the Parallel Boost Graph Library [10], [7] and the Graph500 benchmark reference implementation [1], this will imply potentially $O(|adj(v)|)$ $bytes$ being communicated, where $adj(v)$ gives adjacencies of vertex v. In fact, the amount of communication is:

$$Comm(v) = \sum_{i \in P} m_i(v) \cdot r \quad bytes \tag{1}$$

Where P is the number of processors, $m_i(v)$ gives the number of adjacents of vertex v that are stored on processor i, and r is the size of the message being sent. Traditional BSP libraries employ aggregation to reduce the number of messages. However, the total number of bytes stays the same.

In many cases, for a large class of graph algorithms, such as breadth-first search, PageRank, k-core decomposition, connected components, strongly-connected components, topological sort, betweenness centrality, triangle counting, etc., the vertex sends the same information to all its neighbors. In such cases, the amount of data sent (in bytes) can be reduced using our approach. The following reduction is applicable to such algorithms. For other cases, we may not observe any reduction in the amount of data, but we will still reduce the number of messages sent. The amount of data communicated in the first case is the lowest possible given the algorithm.

Using a hierarchical approach, the vertex may only need to send the result of its computation once for every *super-edge* instead of once for every edge. Therefore, the amount of communication for a hierarchical graph can be given by:

$$Comm^H(v) = \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \quad bytes \tag{2}$$

This effectively means that for any vertex the hierarchical approach performs the least amount of communication possible (both in number of messages and total size of communication) for a given partitioning strategy, as it sends only a single result to each processor that stores any of the vertex's neighbors. Any lower communication can not guarantee the correctness of a general graph algorithm, without changing it.

The total bytes communicated across the system in the hierarchical approach for a graph G can then be given by:

$$\begin{aligned} Comm^H(G) &= \sum_{v \in V} Comm^H(v) \\ &= \sum_{v \in V} \sum_{i \in P} \begin{cases} 1 & : m_i(v) \geq 1 \\ 0 & : m_i(v) = 0 \end{cases} \cdot r \quad bytes \end{aligned} \tag{3}$$

This gives the upper bound of $O(min(V \cdot P, E))$. However, we note that the worst-case upper bound is equivalent to the case of a dense graph, where every vertex is connected to every other vertex and the communication for the traditional level-synchronous approach would have otherwise been $O(V^2)$. For sparser graphs, Equation 3 gives a more accurate estimate. We show empirical evaluation of this communication reduction in Section V-C.

B. Space Overhead

Creating a hierarchical graph adds space overhead compared to the traditional flat graph approach. The hierarchical graph G_1 uses a single vertex corresponding to each partition of the base-graph G_0, which requires $O(p)$ space overall, assuming p partitions in G_0, one per processor. Further, the number of edges in G_1 correspond to the number of partitions in G_0 that have edges between them, with one super-edge per pair of neighboring partitions in G_0. In the worst-case, there can be $O(p^2)$ super-edges in G_1 corresponding to a complete graph in G_0. Therefore the bound on the size of G_1 is $O(p^2)$ for the entire graph, or $O(p)$ overhead per-processor for using the hierarchy. Note that the metadata on super-edges does not contribute to overhead, as it replaces the deleted inter-partition edges from G_0, keeping the total size constant in this regard.

For the hub-hierarchy, every processor stores metadata for any hub vertices that have an edge to a vertex on that processor. This implies $O(|hubs|)$ storage per processor, only if there is an edge to that hub from that processor. However, since the number of such hubs is generally small, the space overhead is

small in practice. Further, our algorithm for creating the hub-hierarchy is parameterized based on a user-specified lower-limit on the size of hub vertices, so the number of vertices treated as hubs can be varied to suit any space constraints on the system.

IV. Implementation

While the hierarchical approach is generally applicable to any distributed memory graph library, for this work, we implemented our approach in the STAPL Graph Library (SGL) [11] to evaluate performance, due to SGL's ease of use and modification, and scalable performance [11], [12] (Section V-A). In this section, we give an overview of SGL and describe the relevant features and extensions needed to support the hierarchical approach. We also show how users can express important graph mining and graph analytics algorithms using our hierarchical paradigm, and thus benefit from our approach.

A. The STAPL Graph Library

SGL is a generic parallel graph library that provides a high-level framework which allows the user to concentrate on parallel graph algorithm development and decouples them from details of the underlying distributed environment. It consists of a parallel graph container (`pGraph`), a collection of parallel graph algorithms to allow users to easily process graphs at scale, and a graph engine that supports level-synchronous and asynchronous execution of algorithms.

The `pGraph` container is a distributed data storage built using the `pContainer` framework (PCF) [21] provided by the Standard Template Adaptive Parallel Library (STAPL) [6]. It provides a shared-object view of graph elements across a distributed-memory machine. The STAPL Runtime System (RTS) and its communication library ARMI(Adaptive Remote Method Invocation) is decoupled from the underlying platform, providing portable performance, thus eliminating the need to modify STAPL applications. The RTS abstracts the physical parallel processing elements into *location*s, components of a parallel machine where each one has a contiguous memory address space and associated execution capabilities (e.g threads). ARMI uses the remote method invocation (RMI) abstraction to allow asynchronous communication on shared objects while hiding the underlying communication layer (e.g MPI, OpenMP).

B. Expressing Graph Algorithms

Graph algorithms in SGL are expressed in a vertex-centric fine-grained manner, and decoupled from parallelism and communication details, as well as from the processing of the graph (e.g. flat or hierarchical). In this section, we show how an example algorithm, breadth-first search (BFS) (Figures 1 and 3) can be expressed in SGL's graph paradigm. Other algorithms such as connected components, k-core, PageRank, community detection, graph coloring, betweenness centrality, pseudo-diameter, etc. can also be expressed in a similar manner. We evaluate these algorithms in Section V-C.

To express an algorithm, the user provides two operators – a *vertex-operator* (Figure 1(a)) which performs the computation of the algorithm on a single vertex and a *neighbor-operator* (Figure 1(b)) that updates the neighbor-vertices of the source vertex with the results of the computation. These two operators are provided to the graph paradigm along with the input graph (Figure 3(a)). The graph paradigm (Figure 4) executes the provided operators on active vertices of the input graph and handles communication, termination-detection of the algorithm, current active vertices, and the execution strategy (level-synchronous or asynchronous). The algorithm terminates when all vertex-operators return false. The user's vertex operators are therefore decoupled from these details and can focus on expression of the algorithm.

```
void BFS(Graph graph, vertex source)
  source.color = GREY;
  graph_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph);
```

(a) Fine-grained BFS

```
void Hierarchical_BFS(Graph graph, HierarchyGraph H,
                      vertex source)
  source.color = GREY;
  hierarchical_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph, H);
```

(b) Hierarchical BFS

Fig. 3. The (b) traditional fine-grained BFS algorithm. The hierarchical version of BFS is shown in (d), where only the paradigm has changed.

```
void graph_paradigm(Graph g, VertexOp wf, NeighborOp uf)
 bool active = true;
 while(active)
   pre_compute(g);
   // apply vertex-operator to each vertex, reduce to
   // find #active vertices. wf returns true (active),
   // or false (otherwise), spawns neighbor-operators.
   active =
    reduce(map(vertex_wf(wf, visitor(uf)), g), logical_or());
   global_fence();
   post_compute(g);
```

Fig. 4. Pseudocode for the graph paradigm.

Breadth-first search (BFS) is an important algorithm due to its widespread direct uses and indirect uses as a part of numerous other algorithms (e.g., betweenness centrality, pseudo-diameter). The overall BFS algorithm results from invoking the *graph_paradigm* (Figure 3(a)) and providing it the operators presented earlier in Figure 1 to obtain the flat (non-hierarchical) BFS. Alternatively, invoking our *hierarchical_paradigm* (Figure 3(b)) with the same operators results in a hierarchical BFS.

Hierarchical Paradigm in SGL. The hierarchical graph paradigm allows the execution of vertex-centric algorithms on hierarchical graphs, as it is a drop-in replacement for the standard graph paradigm, as seen in Figure 3(b), such that existing vertex and neighbor operators need not be modified to take advantage of the hierarchy, or even be aware of it. Different graph algorithms can take advantage of the hierarchical paradigm simply by swapping the call to *graph_paradigm* with *hierarchical_paradigm* and providing it the hierarchical graph.

V. Experimental Evaluation

In this section, we evaluate the performance of our hierarchical approach as compared with the traditional (flat) approach for a set of important graph mining and graph analytics algorithms. We also compare our base-line performance with other graph libraries.

Our experiments were run on three platforms – a Cray XE6 machine with 153,216 cores at the National Energy Research Scientific Computing Center (Hopper), an IBM Blue Gene/Q with 393,216 cores and a smaller Cray XE6m machine with 576 cores available to us. Experiments were run on the Graph 500 benchmark inputs [1], a benchmark for data-intensive and graph applications, as well as other real-world graphs available to us. Results reported are averaged over 32 runs with a high confidence interval. We used the default partitions specified in the input graph files, since partitioners such as ParMETIS are ineffective in partitioning graphs such as Twitter or Kronecker (Graph500 input). Our code was compiled using the GCC 4.8.3 compiler.

A. Comparisons: SGL and Other Libraries

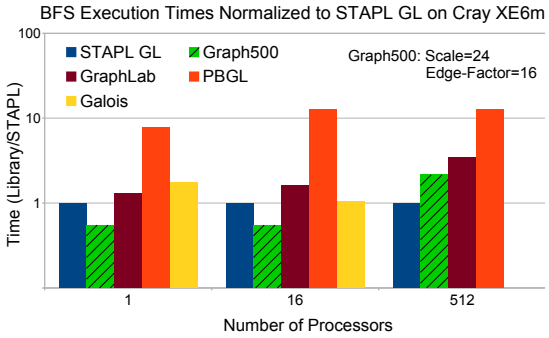


Fig. 5. Execution times of Graph500 on various graph libraries normalized to SGL on CRAY XE6m. Shared-memory libraries shown to 16 cores.

We compare SGL's base (non-hierarchical) implementation, with existing graph libraries for the Graph500 benchmark to establish a base-line for our performance results to follow. These include various popular distributed and shared-memory graph libraries, including the Graph500 benchmark implementation in MPI, Parallel Boost Graph Library (PBGL), Galois, and GraphLab/PowerGraph.

Figure 5 shows their execution times normalized to SGL's base (non-hierarchical) implementation for different processor-counts on a Cray XE6m with 576 cores. SGL's base implementation, using standard techniques such as combiners and aggregators, performs similarly to existing graph libraries, though it is more scalable than comparable distributed-memory libraries such as PBGL and GraphLab, and comparable to shared-memory ones such as Galois [13].

The Graph500 benchmark implementation is initially 1.9x faster than SGL on 16 cores due to low overhead, as the implementation uses arrays of integers to represent their graphs, whereas SGL has a generic graph container. However, SGL scales better in distributed-memory, where at 512 cores SGL is 2.2x faster than the benchmark. There exists an implementation of the benchmark [5] that is 2-3x faster than our baseline, however, this too is not a general-purpose library, but a benchmark-specific implementation that can tradeoff genericity and programmability for targeted performance. GraphLab/PowerGraph [16], [9] is a popular graph processing framework that allows asynchronous and bulk-synchronous computations, similar to SGL. However, it exposes

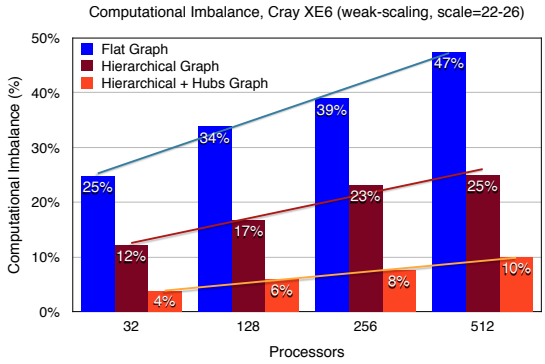


Fig. 6. Comparison of work-imbalance (number of edges) for flat and hierarchical graphs for Graph500 inputs (weak-scaling) on a CRAY XE6m, along with trendlines showing growth rates.

users to low-level details of parallelism such as memory consistency and concurrency, as they have to choose a consistency model for their application and understand its implications. PowerGraph includes optimizations for processing small-world scale-free graphs such as web-graphs. PBGL [10], [7] is a distributed graph library that uses ghost-vertices for communication, which may limit performance at scale. A detailed comparison of SGL with these libraries and others was shown in [11].

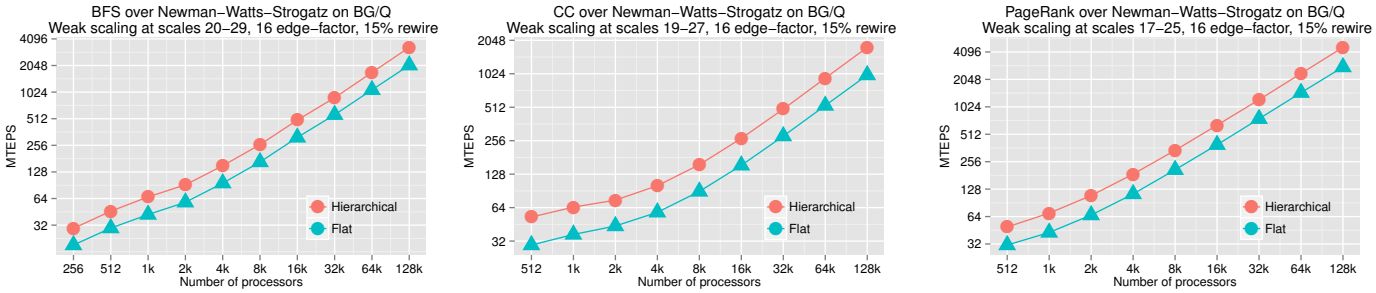
B. Improving Work Imbalance

As the number of processors increases, the graph partition can become imbalanced in the amount of edges each processor has to process. This is further aggravated by scale-free graphs where the hub-size also increases drastically with the size of the graph. This results in the processor storing a high-degree vertex performing more work processing its outgoing edges, which negatively affects scalability and performance. Our hierarchical approach alleviates this by reducing the number of inter-partition edges for each vertex to a single super-edge, and distributing the work of applying updates more evenly as the updates are now applied on the target location instead of the high-degree source.

We show the effect of our hierarchical approach on the work imbalance across processors for different input sizes of the Graph500 input graph at varying processor-counts in Figure 6. The flat partitioning has severe work imbalance which gets worse as the problem and machine size is scaled. The locality-based hierarchical approach is able to reduce this imbalance significantly, while also reducing the rate at which the imbalance grows (demonstrated by the diverging trendlines). The locality-based hierarchical approach in combination with the hubs-based hierarchy further reduces the imbalance significantly and slows down the growth even more. This, as we will observe in the next section, improves scalability and performance through a better load-balanced execution, even while being decoupled from the algorithm.

C. Applications

In this section, we first demonstrate the effectiveness of our approach at scale ($12,000+$ and $130,000+$ cores on Figures 7, 8) on two different large-scale machines, and then dive down to lower core-counts to observe trends and explain



(a) (b) (c)

Fig. 7. Throughput of (a) BFS, (b) connected components and (c) PageRank on Watts-Strogatz input on BGQ, **131,072 cores**. At scale, the hierarchical approach has an improvement of 1.59x, 1.78x and 1.64x over the flat approach, respectively.

our results. The performance benefit of our approach depends on the computation/communication cost ratio. For instances where the communication becomes a bottleneck, our approach benefits greatly. This is dependent on three things: the input graph, the algorithm and the system. A denser graph is harder to partition effectively, and therefore will result in cross-edges with a higher probability, resulting in higher communication costs. On the other hand, algorithms such as betweenness centrality perform heavier computation than algorithms such as breadth-first search, and can therefore hide the communication overhead better, while some systems, such as the IBM Blue Gene/Q, have slower processors and faster networks to achieve the same effect. Figure 7 shows the performance of various algorithms on a Watts-Strogatz small-world network on up to **131,072 cores**. At scale, the hierarchical approach is able to see improvements of $1.59\times$ to $1.78\times$ over the traditional flat algorithm. Our experiments are designed to evaluate our approach on a wide range of important graph algorithms that are representative in their class, as well as different systems.

Performance at Scale. We ran breadth-first search (BFS), connected components (CC), PageRank (PR) and k-core decomposition (KC) algorithms at scale on **12,288 cores** on a Cray XE6 machine to demonstrate the performance benefits of the hierarchical approach at scale. Our results (Figure 8) show a $2.35\times$ performance improvement for breadth-first search, a $7.26\times$ to $8.54\times$ improvement for connected components, a $3.6\times$ to $3.95\times$ improvement for PageRank, and a $4.43\times$ to $5.84\times$ improvement for k-core decomposition. We attribute these improvements to lower communication and better load-balance provided by the hierarchical approach, which improves scalability, as we will demonstrate in this section. Compared to the IBM Blue Gene/Q system, the performance benefits on the Cray XE6 are more substantial using our approach. This is due to the Blue Gene system having slower cores and a faster network, which lowers the computation/communication ratio compared to the Cray XE6. When the ratio is higher (due to a slower interconnect, for example), our approach yields better benefits.

1) Fundamental Algorithms: **Graph500 BFS.** We evaluated the Graph500 benchmark application that simulates data-intensive HPC workloads. The benchmark performs breadth-first traversals of the input graph from multiple source vertices. The Graph 500 input graph simulates social networks and web-graphs and exhibits small-world scale-free behaviour, i.e.,

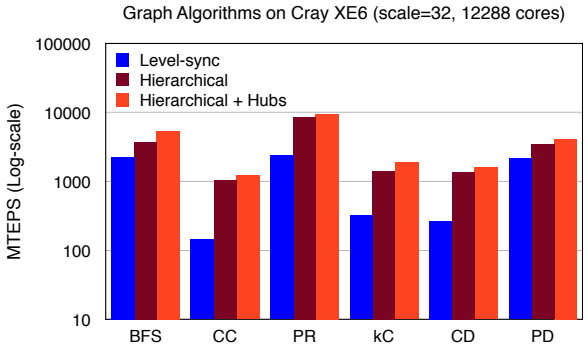


Fig. 8. Scalability (Throughput, log-scale) of various algorithms using hierarchical approach. Graph500 input graph with 4 billion vertices and 64 billion edges at **12,288 cores**.

it has a short diameter ($<$ 16 hops) and vertices with very high out-degrees. These high out-degree vertices (hubs) cause scalability bottlenecks due to communication.

Figure 9(a) compares the number of bytes communicated in the base-line (flat) approach with that in our hierarchical approach. As can be observed, the hierarchical approach is able to significantly reduce the number of bytes sent over the network by $2.7\times$ to $3.3\times$. This directly translates to the performance of the algorithm (Figure 9(b)), where we observe a $1.8\times$ to $2.1\times$ improvement over the base algorithm. For example, at 512 cores, BFS on the Graph 500 input graph communicated (sent/received) 33.87 GB of data across the system in the base (flat) case. This was reduced to 12.56 GB for our hierarchical approach and then further to 10.3 GB for the hierarchical with hubs approach. With the network sending only a third of the data, performance improved.

Connected Components. Connected components [15] (CC) has a heavier communication pattern than breadth-first traversal, and therefore, we expect our hierarchical strategy to provide a higher speedup versus the traditional paradigm. Figure 10 shows this to be the case. As we increase in scale, the benefits of our approach become more evident. At 512 cores, the total bytes communicated was reduced from 111.9 GB in the base-case to 68.91 GB for our hierarchical approach, to 44.1 GB for the hierarchical with hubs approach, a reduction of $1.6\times$ to $2.6\times$ respectively (Figure 10(a)), providing a $2.54\times$ to $3.38\times$ speedup over the base-case (Figure 10(b)).

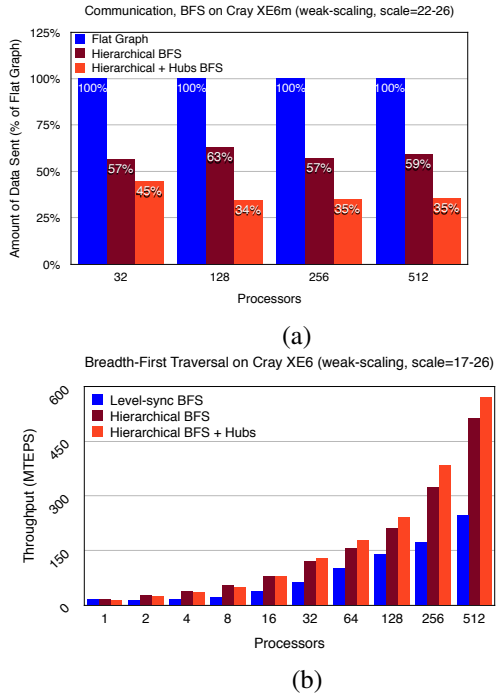


Fig. 9. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of BFS on Graph500 benchmark.

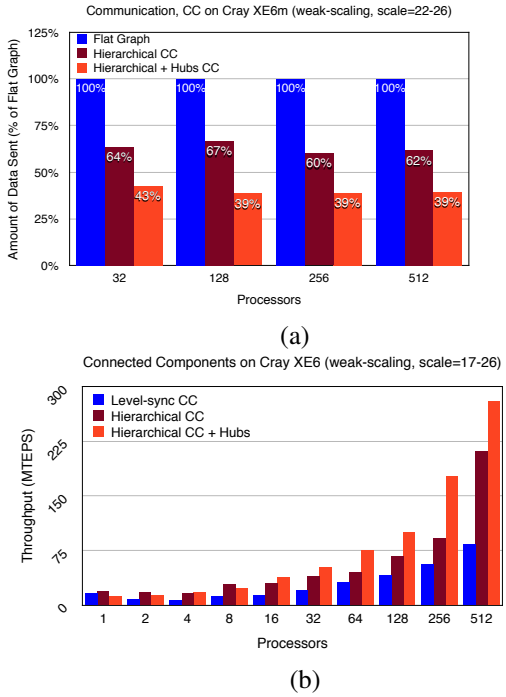


Fig. 10. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of Connected Components on Graph500 input.

2) Graph Mining Algorithms: **PageRank.** PageRank [4], [18] is an important algorithm used to rank web-pages on the internet in order of relative importance. As an example of an iterative random walk algorithm, each vertex calculates its rank in iteration i based on the ranks of its neighbors in iteration $i-1$ and then sends out new ranks to its neighbors for the next iteration. The algorithm terminates when either a certain number of iterations have been reached or the ranks have converged.

PageRank exhibits even heavier communication than connected components, due to all vertices being active (and communicating over all edges) in every iteration of the PageRank algorithm, making it a worst-case scenario for communication. Our evaluation of this algorithm in Figure 11 shows a communication reduction of $1.36\times$ to $2.18\times$ (from 711 GB for flat to 521 GB for hierarchical and 326 GB for hierarchical with hubs), corresponding to a $2.54\times$ to $2.73\times$ speedup over the base-case at 512 cores, which shows the worst-case communication is substantially improved.

k-core Decomposition. A k-core of a graph G is a maximal connected sub-graph of G in which all vertices have degree at least k. The k-core algorithm is widely used to study clustering and evolution of social networks [2]. It is also used to reduce input graphs to more manageable sizes while maintaining their core-structure. The typical parallel algorithm iteratively deletes vertices with degree less than k until only vertices with degree greater than or equal to k exist. k-core has a different communication pattern than either of BFS, CC or PageRank. Here too, the hierarchical approach is able to provide benefits over the base case (Figure 12). At 512 cores, our approach is able to reduce the total bytes communicated across the system from 48.8 GB for the base-case to 21.4 GB for the locality-based hierarchy to 11.2 GB for the hierarchy with hubs, a reduction of $2.3\times$ to $4.3\times$, leading to a speedup of $1.2\times$ to $1.9\times$.

Community Detection. Community Detection is an important application that is widely used to detect groups or clusters in social networks. We use a modularity maximization algorithm to label vertices to their assigned communities. The application iteratively assigns the most frequently occuring label in the local neighborhood of each vertex, until the global quality of the labeling can no longer be improved. Figure 13 shows the improvement in performance obtained by using our hierarchical approach. We note that while the locality-based hierarchy provides a significant improvement in performance, the addition of the hub-based hierarchy does not further improve the performance appreciably. This is due to the fact that, as described in Section II-B, the hub-based hierarchy relies on reducing updates on hubs to a single value on each location. However, for community detection the labels of the entire neighborhood is needed, leading to a minimal reduction for the hub-hierarchy.

3) Application of Traversals: **Betweenness Centrality.** Betweenness Centrality is an important graph-mining algorithm that is used to identify vertices with large influence in a network. For example, it is used to understand the social influence of users on Facebook and Twitter. Our implementation uses Brandes' algorithm [3], which performs forward and backward traversals of the graph to compute the number of shortest paths passing through each vertex. This application directly benefits from our hierarchical approach, as seen in Figure 13. Due to the large number of traversals involved, the actual saving in time is significant. For example, on 512 cores, the execution time of the application was reduced from $1,635.93$ seconds to $1,094.63$ seconds, an improvement of 50%.

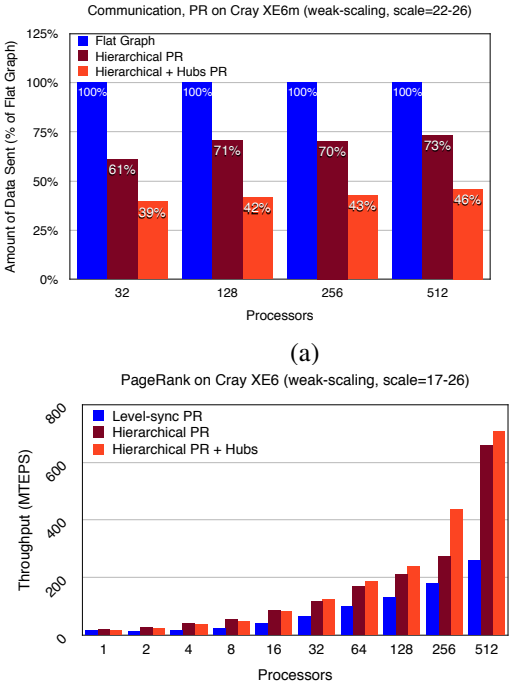


Fig. 11. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of PageRank on the Graph500 input graph.

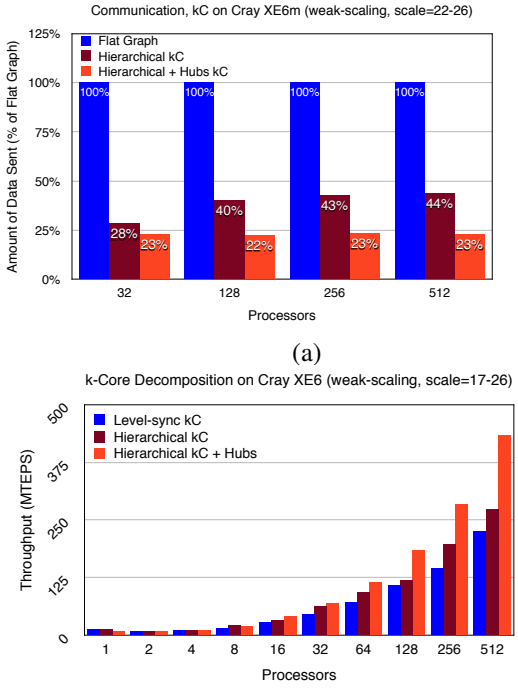


Fig. 12. Communication reduction using hierarchical approach and (b) Scalability (Throughput) of k-core on Graph500 input.

Pseudo-Diameter. Pseudo-Diameter is a graph metric used in network analysis to understand the structure of networks. It iteratively traverses the graph from a source, and selects the farthest vertex as the source for the next traversal, until the distance from a source to its farthest vertex can not be increased. Being a direct application of multiple traversals, improvements (Figure 13) mirror those in BFS.

4) Other Graphs: We also evaluate our approach on the extreme cases of Erdos-Renyi graphs, toroidal meshes and the real-world Twitter social-network graph. A 2D toroidal mesh is the worst-case scenario for our hierarchical approach, as the maximum out-degree of any vertex in the mesh is 4, and when partitioned correctly, the number of cut-edges is minimized. We evaluated our approach on this graph to show that even though we do not expect to improve performance, we add no overhead for such cases either. Figure 14 shows the performance of a BFS traversal of a toroidal mesh with 16 million vertices and 64 million edges, where the overhead of using hierarchies is less than 3%.

On the other hand, an Erdos-Renyi random graph [8] is the best-case for our approach. An Erdos-Renyi network tries to connect each vertex of a graph with every other vertex with a probability p ($p = 1$ leads to a complete graph). We generate an Erdos-Renyi network of 1 million vertices and 5.5 billion edges (a probability $p = .5\%$) to show the benefit of our hierarchical approach when the connectivity of the input graph is high. This is shown in Figure 14, where the hierarchical approach is $42.5\times$ faster.

To evaluate how the performance improvement varies with graph connectivity, we measured the performance improvement of our approach on a Watts-Strogatz random network for

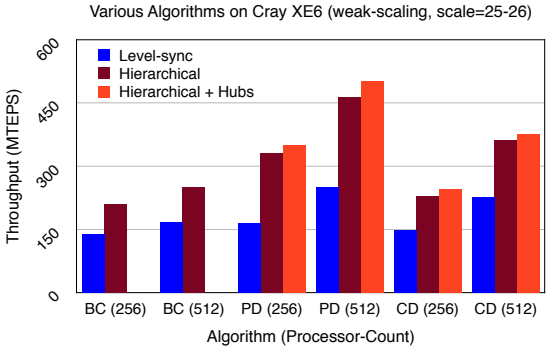


Fig. 13. Throughput of Betweenness Centrality (BC), Pseudo-Diameter (PD), and Community Detection (CD) on Graph500.

varying rewire probabilities. The Watts-Strogatz model [23] produces random networks with the small-world properties such as short average path lengths and a high degree of clustering. The model starts with a regular ring lattice with every vertex connected to its k nearest neighbors on either side, forming a ring-like structure. Thereafter, for each vertex, each of its edges is 'rewired' with a probability β, such that the target of the edge is selected with uniform probability from the remaining vertices, avoiding self-loops and duplicate edges. By varying β, one can generate graphs that are ring-like and do not exhibit small-world properties (for small values of β), to small-world networks with short path-lengths and high degree of clustering (as β approaches 1).

Figure 15 plots the speedup of our approach over the baseline as the rewire-probability is varied from 0 to 1 for various processor counts. While our approach has a 25% overhead

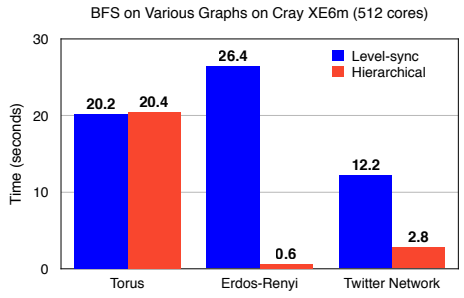


Fig. 14. Running times of BFS on various input graphs using flat and hierarchical approaches.

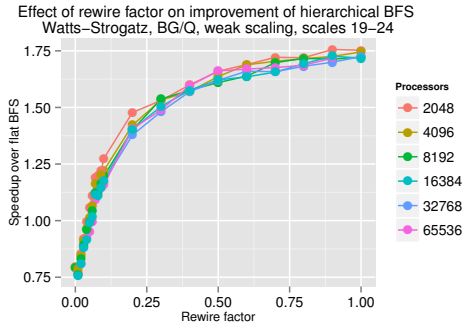


Fig. 15. Improvement of the hierarchical approach on BFS for varying rewiring-probability of a Watts-Strogatz graph.

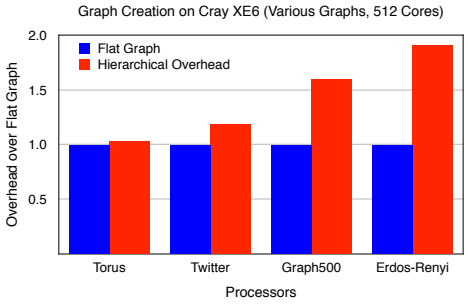


Fig. 16. Comparison of total construction times of flat and hierarchical graphs for various inputs.

on the BG/Q machine for rewire-factor $\leq 3\%$, it shows an improvement over the base-line for all other rewire-factors ($3\% - 100\%$). The initial overhead is due to the graph being well-partitioned with extremely few cut-edges. In this case, there is not much communication to reduce, and we just measure the overhead of going through the hierarchy. It is higher for the BG/Q machine than the Cray XE6 (3% for torus in Figure 14, and we measured 1.5% for Watts-Strogatz with 0% rewiring) as the processors are much slower and the network is extremely fast, so the effect is more pronounced. This can be eliminated by profiling the system using this example to compute the number of cross-edges beyond which the hierarchical approach should be used. As the algorithms remain the same, this can be done cheaply at runtime.

We can observe that the improvement over the base-line increases as the small-world behavior increases. Small-world graphs are harder to partition well and produce a large number of cross-partition edges, which result in heavy communication, thus limiting performance. Our approach alleviates this, improving performance. This is evident even in machines where the computation/communication ratio is low (i.e. better network, slower processors), such as the BG/Q. For machines with (comparatively) slower networks and faster processors, the improvement is even better, as observed in experiments on the Cray XE6 (Figure 8 and others).

Finally, we also evaluate our approach on the Twitter social network from 2010. This network has 65 million vertices and 1.2 billion edges, and has large hub vertices corresponding to popular users who have many followers. The presence of large hubs leads to poor scalability, which can be effectively addressed by using the hierarchical approach. As a result of communication reduction from 21 GB to 8.2 GB and a better load-balance, our approach provides a speedup of $4.36\times$ (Figure 14) over the flat approach.

D. Overhead of Hierarchy Creation

In order to run hierarchical algorithms, the input graph needs to be converted to its hierarchical representation on the machine. This is a one-time process following graph construction, after which multiple algorithms can take advantage of the hierarchy. Figure 16 compares the time to construct the flat graph to the time to construct the hierarchical graph. The time to generate the hierarchical graph includes the time to create the flat graph. Hierarchical graph creation overhead is proportional to the number of cross-processor edges in the graph. Therefore, for graphs with few cross-processor edges, such as a toroidal mesh, the hierarchy construction adds negligible overhead, while for graphs with a large number of cross-processor edges, such as a dense Erdos-Renyi graph, the hierarchy creation overhead is larger (Figure 16). Correspondingly, the benefit of the hierarchy is also significantly larger for the Erdos-Renyi graph vs. the torus (Figure 14). As an example, on 512 cores, the graph construction time for 67 million vertices and 2.1 billion edges was 8.5 seconds, while creating the hierarchy added 5.2 seconds to that time. A comparison showing the running time of a *single run* of various algorithms on hierarchical and flat graphs is shown in Figure 17, along with the time required to create the hierarchies, to show the overhead. For example, a PageRank algorithm using the traditional approach takes 68.7 seconds, while a PageRank using the hierarchical approach uses 39.7 seconds. To amortize the cost of hierarchy creation, BFS would need to be run 4 times, however, in many use-cases, such as betweenness centrality and pseudo-diameter, BFS is usually executed multiple times from different sources, allowing the hierarchy creation to be ammortized, even for a single execution of the algorithm (Figure 17). Other algorithms, such as community detection, PageRank, and k-core decomposition also observe significant overall benefits even in a *single* execution, including the cost of hierarchy creation, as shown in Figure 17. We note that the hierarchy is algorithm-agnostic and only depends on graph structure, and not the metadata (vertex/edge properties), and can therefore be used with different algorithms once created.

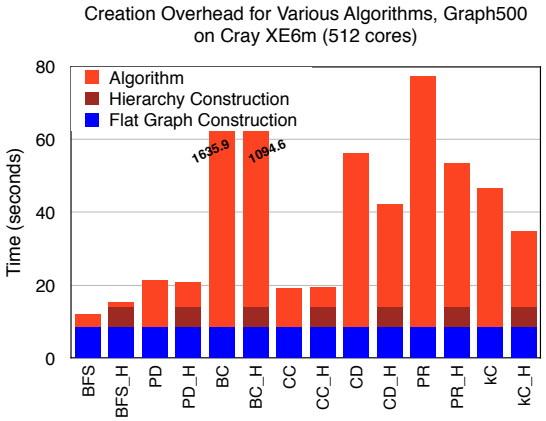


Fig. 17. Comparison of graph construction and algorithm execution times of flat and hierarchical graphs for Graph500 input.

VI. Related Work

To enable graph algorithms to scale, various different approaches have been adopted. Many implementations [1], [17], including our baseline implementation, aggregate messages being sent to a processor in order to reduce the number of messages sent. However, the *total number of bytes sent is not reduced*, but merely concatenated to form larger messages. Pregel [17] also allows users to specify a *combiner* that may be used to reduce multiple incoming messages to a given vertex to a single value per processor. However, this is not applicable to all algorithms and also not guaranteed to be executed in Pregel. Moreover, combining can only address reduction of fan-in, not high out-degree/fan-out vertices.

Another approach is to use ghost vertices to facilitate communication, which cache values of neighboring vertices stored on other processors. However, such vertices need to be coherent with their original vertices, and do not scale well in practice, due to storage and communication overhead of maintaining the ghosts, as shown in [9]. This approach is used by the Parallel Boost Graph Library [10], which we compare against in our experiments.

There have also been methods that propose a 2-dimensional partitioning of the graph and its edges to achieve better scalability by reducing communication cost [5]. To use such an approach, the algorithms need to be rewritten to account for a distributed edge-list and made aware of the underlying data-distribution, making this method difficult to use in practice. This method also does not reduce the message size, but does distribute the computation more evenly across partitions than 1-D partitioning. However, 2-D partitioning does not consider the locality of the target vertices of the edges being partitioned, and thus may in fact lead to more hops for the message to reach its destination, for example, one hop to get to the processor where the edge is stored, and a second hop to get to where the target-vertex of that edge is stored (as edges are not co-located in 2-D partitioning), generating extra communication. Our approach produces co-located edges in addition to lowering the computational imbalance, without user intervention.

Some approaches [9], [14], [19], [20], [24] have also proposed splitting hubs across multiple processors. PowerGraph [9], a version of GraphLab [16] designed for processing scale-free graphs uses the concept of *vertex-cuts/vertex mirroring* to split hub vertices across partitions. However, as mentioned in [9], due to maintaining the vertex-splits, their ghost vertices need to be kept synchronized across all machines it spans. This is addressed in PowerGraph by minimizing the number of processors across which the hubs are split to lower the synchronization costs. However, this limits the available parallelism for processing the hubs. Our approach is not limited by this, since we do not have ghost vertices and do not need to maintain state information across partitions. We compare our base-line with the latest available version of PowerGraph in Section V-A. Pregel+ [24] also uses mirroring, which reduces communication by partitioning the edges of high-degree (hub) vertices. However, this does not have much benefit in their published results, and performance actually degrades when the number of hubs is large. Our hierarchical technique, on the other hand, is applied to all vertices and consistently shows benefits, even when used for low-degree vertices.

On the other hand, [19], [20] create copies of the hubs on all processors, which may be wasteful if the hubs do not have adjacent vertices on all processors, and increases storage and communication overhead. In these cases, the hubs need to be coherent across the processors, which leads to extra communication, and a computational imbalance may still remain. This approach of splitting hubs also does not address non-hub vertices due to the overheads involved in splitting vertices and the storage requirements for replicating data. The algorithm-writers too need to be aware of the presence of such hub-representatives, and the communication between them, requiring the user to re-write their algorithm. We improve upon this work with our hubs approach which allows for local reduction of updates for hub vertices on processors where they are needed, while alleviating the need to keep them coherent. Further, it is transparent to the algorithm. This is in addition to our locality based hierarchy, and is explained in Section II-B.

The authors in [24] propose a request-response paradigm where a vertex can request data from another random vertex and it will be available in the next superstep. For algorithms that need this pattern, it can reduce the amount of back traffic by not sending duplicate values to the same processor. However, this is only applicable for a restricted set of algorithms explored in their paper, and can not be applied to widely used algorithms such as traversals, PageRank, etc. It also requires the algorithm to be modified to incorporate this paradigm for the cases where it is applicable. Their results show a modest improvement in performance for such cases.

Our hierarchical approach operates at a semantic level, allowing us to reduce the information sent over the network for both outgoing and incoming edges, thereby improving scalability. This is done for all vertices that have cross-processor edges, not only for hub vertices. Since information is not replicated, and due to using an asynchronous push-model, we do not incur significant storage or communication overheads endemic to previous approaches. Crucially, due to operating at a semantic level, the algorithm itself remains unaware of the hierarchy, allowing the reuse of existing fine-grained graph algorithms as-is with the hierarchy.

VII. Conclusion

We presented a technique for hierarchical algorithmic coarsening that reduces the number of bytes communicated in a distributed system, improving scalability and performance of graph algorithms, while allowing the reuse of existing fine-grained graph algorithms. We implemented this technique in the SGL framework, and evaluated it on two large-scale systems ($12,000+$ cores and $130,000+$ cores, Figures 7, 8) on various important graph analytics and graph mining algorithms with benchmark and real-world graphs, observing significant speedups of $2.5\times$ to $8\times$ at scale, without significant penalty in cases where performance was not improved.

VIII. Acknowledgments

We would like to thank Ioannis Papadopoulos for helping with our initial design and optimizations in our runtime-system. We would also like to thank our anonymous reviewers. This research is supported in part by NSF awards CCF 0702765, CNS-0551685, CCF-0833199, CCF-1439145, CCF-1423111, CCF-0830753, IIS-0917266, by DOE awards DE-AC02-06CH11357, DE-NA0002376, B575363, by Samsung, IBM, Intel, and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This research used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Dept. of Energy under Contract No. DE-AC02-05CH11231.

References

[1] The graph 500 list. http://www.graph500.org, 2013.

[2] J. I. Alvarez-hamelin, A. Barrat, and A. Vespignani. Large scale networks fingerprinting and visualization using the k-core decomposition. In *Adv. in Neural Inf. Proc. Syst. 18*, pp. 41–50. MIT Press, 2006.

[3] U. Brandes. A faster algorithm for betweenness centrality. *J. of Math. Sociology*, pp. 163–177, 2001.

[4] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Sys.*, pp. 107–117, 1998.

[5] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. of Intl. Conf. High Perf. Comp., Networking, Storage and Anal.*, SC '11, pp. 1–12, 2011.

[6] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. STAPL: Standard Template Adaptive Parallel Library. In *Proc. Haifa Exp. Sys. Conf. (SYSTOR)*, pp. 1–10, 2010.

[7] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proc. Symp. on Princ. and Prac. of Par. Prog.*, PPoPP '13, pp. 289–290.

[8] P. Erdos, and A. Renyi. On Random Graphs. I In *Publ. Mathematicae*, pp. 290–297, 1959.

[9] J. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. *Proc. OSDI*, pp. 17–30, 2012.

[10] D. Gregor, and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Par. Object-Oriented Sci. Comp.*, 2005.

[11] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In *Lang. and Compilers for Par. Comp.*, LNCS, pp. 46–60. Springer, 2012.

[12] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. KLA: A new algorithmic paradigm for parallel graph computations. In *Proc. Intl. Conf. on Par. Arch. and Comp. Techniques*, PACT '14, pp. 27–38, Canada, 2014. ACM.

[13] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered and unordered algorithms for parallel breadth first search. In *Proc. Intl. Conf. on Par. Arch. and Comp. Techniques*, PACT '10, pp. 539–540, 2010.

[14] I. Hoque, and I. Gupta. LFGraph: Simple and Fast Distributed Graph Analytics. In *Proc. Conf. on Timely Results in O.S.*, pp. 1–17, 2013.

[15] U. Kang, C. E. Tsourakakis, and C. Faloutsos. PEGASUS: A Peta-Scale Graph Mining System Implementation and Observations. In *Proc. IEEE International Conference on Data Mining*, ICDM '09, pp. 229–238. 2009

[16] 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. *VLDB*, pp. 716–727, 2012.

[17] 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 *Proc. Intl. Conf. on Mgmt. of data*, SIGMOD, pp. 135–146, 2010.

[18] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking. 1998.

[19] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In SC '10, pp. 1–11, 2010.

[20] R. Pearce, M. Gokhale, and N. M. Amato. Faster Parallel Traversal of Scale Free Graphs at Extreme Scale with Vertex Delegates. In *Proc. Intl. Conf. High Perf. Comp., Networking, Storage and Anal.*, SC '14.

[21] G. Tanase, A. Buss, A. Fidel, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, N. Thomas, X. Xu, N. Mourad, J. Vu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Container Framework. In *Proc. Symp. on Princ. and Prac. of Par. Prog.*, PPoPP, pp. 235–246, 2011.

[22] L. Valiant. Bridging model for parallel computation. *Comm. ACM*, pp. 103–111, 1990.

[23] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, pp. 440–442, 1998.

[24] D. Yan, J. Cheng, Y. Lu, and W. Ng. Effective Techniques for Message Reduction and Load Balancing in Distributed Graph Computation *WWW* 2015.