KLA: A New Algorithmic Paradigm for Parallel Graph Computations

Harshvardhan Adam Fidel Nancy M. Amato Lawrence Rauchwerger

Parasol Laboratory
Deptartment of Computer Science and Engineering
Texas A&M University
{ananvay, fidel, amato, rwerger}@cse.tamu.edu

ABSTRACT

This paper proposes a new algorithmic paradigm – k-level asynchronous (KLA) – that bridges level-synchronous and asynchronous paradigms for processing graphs. The KLA paradigm enables the level of asynchrony in parallel graph algorithms to be parametrically varied from none (level-synchronous) to full (asynchronous). The motivation is to improve execution times through an appropriate trade-off between the use of fewer, but more expensive global synchronizations, as in level-synchronous algorithms, and more, but less expensive local synchronizations (and perhaps also redundant work), as in asynchronous algorithms. We show how common patterns in graph algorithms can be expressed in the KLA pardigm and provide techniques for determining k, the number of asynchronous steps allowed between global synchronizations. Results of an implementation of KLA in the STAPL Graph Library show excellent scalability on up to 96K cores and improvements of 10x or more over level-synchronous and asynchronous versions for graph algorithms such as breadth-first search, PageRank, k-core decomposition and others on certain classes of real-world graphs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel Programming*; D.2.13 [**Software Engineering**]: Reusable Software—*Reusable Libraries*

Keywords

Parallel Algorithms; Asynchronous Graph Algorithms; Graph Analytics; Big Data; Distributed Computing.

1. INTRODUCTION

Traversals – wherein a computation proceeds from one vertex to another along the edges of a graph – are an important type of graph algorithm, as they form the backbone of several other important graph algorithms (e.g., shortest paths, centrality metrics, connected components). Improving the performance of traversals therefore benefits all algorithms dependent on them. Despite receiving a great deal of attention from many researchers for several decades [26, 13, 5, 21, 27, 8, 17, 15, 19], traversal-based computations remain notoriously difficult to parallelize effectively. Similarly, random walks (such as PageRank) are widely used on web-scale graphs and social networks for link-analysis [7, 23], while algorithms such as k-core decomposition are used to study clustering and evolution of social networks [4]. Such algorithms need effective parallelization as well.

Parallel graph algorithms are currently expressed in level-synchronous or asynchronous paradigms. Level-synchronous paradigms [26, 8] iteratively process vertices of a graph level by level. This model guarantees the current level's computation to have completed before starting the next one through the use of global synchronizations at the end of each level. Level-synchronous algorithms tend to perform well when the number of levels is small, but suffer from poor scalability when the number of levels is large. Bulk synchronous parallel (BSP) algorithms [30] can naturally be expressed in this paradigm. The asynchronous paradigm replaces global synchronizations with point-to-point synchronizations, which can increase the degree of parallelism, but which may also require the completion of redundant work. For example, an asynchronous breadth-first search (BFS) may re-visit vertices multiple times as shorter paths are discovered [24]. Choosing the right paradigm depends on the system, input graph, and algorithm. This implies different implementations and optimizations for algorithms, with no easy way to switch between them.

In this paper, we propose a new paradigm, k-level asynchronous (KLA), that allows parametric control of asynchrony ranging from completely asynchronous execution to partially asynchronous execution to level-synchronous execution. Partial asynchrony is achieved by generalizing the BSP model to allow each superstep to process up to k levels of the algorithm asynchronously.

We implement the KLA paradigm using the STAPL Graph Library (SGL) [14] and evaluate its performance and scalability up to 98,000 cores on a Cray XE6 machine for important classes of graph algorithms including traversals (e.g., BFS, single-source shortest paths, connected components), random walks (e.g., PageRank) and k-core decomposition. Our experimental studies show that KLA improves the performance of these algorithms on some graphs by a factor of 10 or more, at scale.

PACT'14, August 24–27, 2014, Edmonton, AB, Canada.

http://dx.doi.org/10.1145/2628071.2628091 .

In summary, our contribution is threefold:

- **k-level async (KLA) algorithmic paradigm.** We propose a novel paradigm to express parallel graph algorithms that unifies asynchronous and level-synchronous paradigms, while allowing parametric control of asynchrony. We also provide a model to estimate the appropriate level of asynchrony for a given input graph, and a strategy to adaptively determine k.
- **Application of KLA to standard graph computations.** We show how to transform common patterns of asynchronous and level-synchronous graph algorithms to the KLA paradigm.
- **Implementation that achieves scalable performance.** Our implementation of KLA in the STAPL Graph Library shows an improvement of 10x or more over level-synchronous and asynchronous paradigms at large scale (up to 98,000 cores) for graph algorithms such as BFS, PageRank and k-core decomposition.

2. THE KLA PARADIGM

k-level async (KLA) is a novel algorithmic paradigm for parallel processing of graphs. The KLA paradigm bridges traditional level-synchronous and asynchronous paradigms by enabling the level of asynchrony to be parametrically varied from none (level synchronous) to full (asynchronous). The motivation is to improve execution time by using an appropriate combination of expensive global synchronizations, as in level-synchronous algorithms, and less expensive local synchronizations (and possibly redundant work), as in asynchronous algorithms.

This paradigm works in phases, similar to the BSP model. However, each phase is allowed to proceed asynchronously up to k steps by creating asynchronous tasks on active vertices. When $k = 1$, KLA processes the graph one level at a time, i.e., in a level-synchronous fashion. Assuming the level-synchronous variant of the algorithm performs d iterations, if $k \geq d$, then KLA is equivalent to the asynchronous paradigm. However, for $1 < k < d$, the algorithm proceeds in $\lceil \frac{d}{k} \rceil$ phases, referred to as *KLA supersteps* (KLA-SS). Each superstep processes up to k levels asynchronously. The KLA paradigm requires that the algorithmic invariant holds at the point of synchronization. However, in between synchronizations, no guarantee is implied.

Figure 1 provides pseudocode for the KLA paradigm. The paradigm takes two fine-grained vertex-centric functions: a *vertex-operator* and a *neighbor-operator*. The *vertex-operator* describes the computation to be performed on each vertex. It processes the vertex on which it is spawned, and is allowed to visit other vertices by applying the *neighbor-operator* on them. The *vertex-operator* returns *true* if the vertex was actively processed, or *false* if the vertex was not active. A vertex is active if its value is *updated* by the *vertex-operator*, or if it visits neighboring vertices. The paradigm also allows users to optionally provide pre- and post-compute methods, which will be called at the begining and end of each KLA-SS. To express an algorithm in the KLA paradigm, users provide the vertex and the neighbor operators.

In the following sub-section, we describe in more detail how an important graph computation, breadth-first search (and therefore its applications such as single-source shortest paths and connected components) can be expressed in the KLA paradigm. Other algorithms, such as k-core and PageRank, can also be expressed in the KLA paradigm by using the appropriate vertex and neighbor operators (Figs. 4,5). This is described in Sec. 3.1, which also analyzes implications of applying KLA to algorithms.

```
void kla_paradigm(Graph graph, VertexOp wf, NeighborOp uf, int k)
  bool active = true;
  int k_current = 1;

  while( active ) {
    pre_compute(graph, k_current);

    // apply the vertex−operator to each vertex, and reduce to
    // find the no. of active vertices. wf returns true (active), or
    // false (otherwise), and spawns neighbor−operators on neighbors.
    active =
      reduce(map(kla_wf(wf, kla_visit(uf, k)), graph), logical_or());
    global_fence();
    k_current += k; // current level is k more than before.

    post_compute(graph, k_current);
  }
```

Figure 1: Pseudocode for the KLA paradigm.

2.1 KLA Breadth-First Search

Breadth-first search is a common graph algorithm, widely used as a backbone to implement many other graph algorithms. Multiple algorithms exist to perform BFS in parallel, including the level-synchronous [26, 8, 19] and asynchronous [15, 24] BFS algorithms. Depending on the input graph and the system, choosing the wrong implementation may provide worse performance. We demonstrate how we can significantly improve performance by bridging these two algorithmic approaches using the KLA paradigm and taking advantage of the tunability.

Figure 2 provides an intuition of how the level-synchronous, asynchronous, and KLA versions of BFS differ. From the user's perspective, the driver of the algorithm (Figure 3(a)) does not change, and neither do the BFS vertex and neighbor operators for the algorithm (Figure 3(b,c)). What does change is the `Visit` logic (Figure 2). This determines conditions to allow the computation to proceed on the neighbor vertex being visited. While the visit logic has been simplified in the figure to make it readable, the BFS code closely corresponds to what a user would write.

A KLA BFS results by providing the generic BFS vertex and neighbor operators (Figure 3(b,c)) to the KLA paradigm. The vertex operator can use the functions provided by the paradigm (Visit, VisitAllNeighbors, VisitNeighborIf) to apply the BFS neighbor-operator to neighboring vertices. The vertex operator returns true if it is active for a vertex, and false otherwise. The algorithm terminates when all vertices are inactive, i.e., when all invocations of the vertex operator return false. The requirements for the KLA neighbor-operator are the same as for an asynchronous BFS – it cannot rely on the order in which the vertex is visited and must be resilient to multiple visits. The correctness of the KLA BFS can be proved using simple induction, as shown below.

LEMMA 2.1. *At the end of the i-th KLA BFS KLA-SS, all vertices at distance $\leq i \cdot k$ from the source have been visited and labeled with the correct distances, and no vertices of distance $> i \cdot k$ from the source have been visited.*

```
Visit(NeighOp update, Vertex u, int k_current)
  bool active = update(u)
  if (active)
    bfs_vertex_op(u);
```

(a) Async Visit
spawn tasks

```
Visit(NeighOp update, Vertex u, int k_current)
  bool active = update(u)
  if (active && k_current % k != 0)
    bfs_vertex_op(u);
```

(b) KLA Visit
spawn tasks to depth k

```
Visit(NeighOp update, Vertex u, int k_current)
  update(u);
```

(c) Level-Sync Visit
don't spawn further tasks

Figure 2: Visitation logic for async, KLA and level-sync behavior of BFS.

```
void BFS(Graph graph, Vertex source, int k)
  source.color = GREY;
  kla_paradigm(bfs_vertex_op(), bfs_neighbor_op(), graph, k);
```

(a) BFS algorithm driver

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

(b) 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;
```

(c) neighbor-operator

Figure 3: The fine-grained BFS algorithm.

PROOF. The proof is by induction on i, the KLA-SS number. When $i = 0$, only the source has been visited and it is trivially true. Next, we assume that at the end of KLA-SS $(i-1)$, all vertices at distance $\leq (i-1) \cdot k$ from source have been visited and correctly labeled with distances, and vertices at a distance $> (i-1) \cdot k$ are unvisited.

At the beginning of KLA-SS i, only vertices at distance $(i-1) \cdot k+1$ from source are active. The vertex-operator in Figure 3(b) is applied to all these active vertices and visits their neighbors (Figure 3(b), line 3) if they are at a distance $\leq i \cdot k$ from the source (Figure 2(b), line 3). During the visit, the distance to source is updated only if it decreases; hence, once the correct minimum distance is computed it will not be further updated. Moreover, since the vertex-operator visits all neighbors of active vertices, all edges in the sub-graph induced by the vertices visited during the ith KLA-SS, i.e., at distance $(i-1) \cdot k < d \leq i \cdot k$ from source, will be processed, implying that at some point during the KLA-SS, the correct distance will be computed for all vertices visited in that KLA-SS. □

COROLLARY 2.2. *KLA BFS will perform* $\lceil \frac{d}{k} \rceil$ *KLA-SSs, where* d *is the number of levels in the graph.*

COROLLARY 2.3. *If* $k = 1$*, then KLA BFS visits the vertices of the graph in a level-synchronous manner.*

2.2 KLA and the Δ-Stepping SSSP Algorithm

A related approach to KLA is the Δ-stepping single-source shortest path [22] (SSSP) algorithm. It processes multiple *light* edges simultaneously in a single superstep, as long as the smallest tentative distance of an edge's source falls between $i\Delta$ and $(i+1)\Delta$, where i is the current superstep. Then, the *heavy* edges of the vertices that were relaxed for the current superstep are processed, ending the superstep. This greedy approach gives preference to lighter-weight edges, which may improve execution time.

The Δ-stepping algorithm differs from KLA in that it centers around using edge-weights as the metric to prioritize the processing of edges, which has no impact on non-SSSP based algorithms. The KLA paradigm works on the structure of the graph, and not algorithm specifics (edge/vertex weights, etc.). This makes it algorithm agnostic and thus applicable to a broader set of algorithms.

While Δ-stepping is beneficial for solving the single-source shortest path and related problems, it is not applicable to algorithms such as k-core or PageRank. On the other hand, k-core, PageRank and Δ-stepping itself may be expressed using the KLA paradigm. Therefore, KLA is a generalization and extension of the Δ-stepping algorithm to other types of computations and metrics. Section 5.6 compares the traditional Δ-stepping algorithm with its KLA variant.

3. APPLYING KLA TO ALGORITHMS

In this section, we demonstrate how to express graph algorithms in the KLA paradigm. We begin by analyzing the asynchronous behavior for a few important graph algorithms. We then present a classification describing how to apply KLA to additional algorithms.

3.1 Analysis of Example Algorithms in KLA

We analyze three important graph algorithms (BFS, k-core and PageRank) that display different asynchronous characteristics and show how the KLA paradigm may be applied to each. For some algorithms, there is no penalty paid as the asynchrony in the algorithm increases, while in other algorithms, increasing the asynchrony leads to a dramatic increase in redundant work, with the algorithm having to correct previously computed values due to out-of-order arrival of messages. The best performance is obtained by balancing the penalty, if any, due to asynchrony with the cost of global synchronizations. In KLA, this is done by limiting the size of the asynchronous sections by introducing global synchronizations at intervals of depth k.

Breadth-First Search and Graph Traversals. Breadth-first search is an important algorithm due to its widespread direct uses and indirect uses as a part of numerous other algorithms (e.g., betweenness centrality, connected components). We show how to implement KLA BFS in section 2.1. Here, we analyze the implications of applying KLA on BFS.

In a completely asynchronous version of BFS, each vertex starts out with its distance from the source vertex initialized to infinity, and the source vertex's distance initialized to

zero. When a vertex receives a message and is updated with a new distance less than its current distance, it propagates this new distance (plus one) to its neighbors. The vertex may receive a shorter distance in the future, which it will also propagate. In this case, the work done by the previous propagation will be redone, resulting in redundant work. However, this scenario cannot be avoided as the vertex is not necessarily aware of the number of messages it is going to receive *a priori*. Because of this, the BFS computation must optimistically propagate every message it receives for a vertex, leading to possible redundant work.

Other graph-traversal algorithms also exhibit similar behavior, along with algorithms dependent on such traversals. This class of algorithms is not aware of the number of incoming messages a vertex will receive *a priori*.

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. 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.

Figure 4 gives the algorithm for k-core as expressed using KLA. In this algorithm, each vertex starts with a count initialized to its degree. A vertex marks itself deleted if its degree count is less than the specified k value. When a vertex is deleted, it invokes a neighbor-operator over all its adjacent vertices to decrement their degree-counts by one (due to the now deleted edges of the deleted vertex), and activate the adjacent vertex, which can now check its count and repeat the process. This purely asynchronous algorithm relies on determining when to trigger propagation of each vertex based on how many messages it has received. That is, a vertex "knows" the number of messages required for the propagation to take place. Moreover, as there are no constraints on the ordering of the updates (because the degree-count is a one-dimensional scalar being decremented), there is no penalty for increasing asynchrony.

Similarly, the parallel topological sort algorithm also exhibits this behavior. Each vertex starts with a count of its in-degree, and only propagates when the vertex becomes a source, i.e., when its in-degree counter becomes zero.

This class of algorithms is aware of the number of incoming messages needed to activate a vertex *a priori* and is unaffected by the order in which the messages are received.

PageRank. PageRank [7, 23] is an important graph algorithm that is 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.

Even though commonly expressed as a level-synchronous algorithm, PageRank can be converted to a KLA form similar to the k-core algorithm (Figure 5). When a vertex receives the ranks of all its neighbors from iteration $i-1$, it can propagate its rank to its neighbors for the next iteration. For this, each vertex must keep a count of how many ranks

```
bool kcore_vertex_op(Vertex v)
  if (v.degree_count < K && !v.deleted) // Propagate if degree <K
    v.deleted = true;
    VisitAllNeighbors(v, kcore_neighbor_op(_1));
    return true;                  // I was Active
  else return false;              // I was Inactive
```

(a) vertex-operator

```
bool kcore_neighbor_op(Vertex u)
  if (!u.deleted)
    u.degree_count--;             // update my degree
    return true;                  // I was updated
  else return false;
```

(b) neighbor-operator

Figure 4: The k-core algorithm

```
bool pagerank_vertex_op(Vertex v)
  // Propagate if ranks from all neighbors received and <20 iterations
  if (v.num_rcvd_ranks[v.iteration] == v.size() && v.iteration < 20)
    v.rank = 0.15/num_vertices + 0.85*v.sum_rcvd_ranks[v.iteration];
    v.iteration++;
    int n = v.neighbors().size();
    VisitAllNeighbors(v, pr_neigh_op(_1, v.rank/n, v.iteration));
    return true;                  // I was Active
  else return false;              // I was Inactive
```

(a) vertex-operator

```
bool pr_neigh_op(Vertex u, double rank, int iteration)
  // update number and rank for given iteration
  u.num_rcvd_ranks[iteration]++;
  u.sum_rcvd_ranks[iteration] += rank;
  return true;                    // I was updated
```

(b) neighbor-operator

Figure 5: The PageRank algorithm.

it has received from the previous iteration. Moreover, in an asynchronous execution, it may also receive ranks for a future iteration before having received all ranks for the $i-1^{th}$ iteration. In such cases, the vertex must buffer this rank in a two-dimensional table, indexed by iteration, for future use. When the vertex propagates and reaches the next iteration, it can use any buffered ranks stored for this new iteration.

As in the case of k-core, each vertex is aware of the number of messages (ranks) it needs to receive to propagate. However, unlike k-core, PageRank also requires ordering information. That is, each vertex must receive all ranks from the previous iteration before processing any ranks from subsequent iterations. Because of the necessary ordering constraint, increasing asynchrony may increase the amount of messages buffered. This is also true for other random walk algorithms that are similar to PageRank, as well as other algorithms like pointer jumping and graph coloring.

3.2 KLA Algorithmic Categories

Based on the characteristics of algorithms discussed in the previous section, we observe that the penalty paid for increasing asynchrony depends on two properties related to the behavior of the algorithm upon a vertex receiving a message from one of its neighbors:

- **Ordering.** Is the algorithm resilient to the ordering of messages from different iterations (i.e., is it algorithmically valid [correct] for messages from iteration $i-1$ to arrive before messages from iteration $i-2$)?

- **Message Count.** Does the algorithm have *a priori* knowledge (on a per-vertex basis) of the number of messages required to activate each vertex?

If the algorithm is resilient to the ordering, it may or may not be aware of the number of messages a vertex needs to receive in order to propagate further. We define propagation to be the necessary condition by which an incoming message along an in-edge triggers a propagation of an update request along all outgoing edges of that vertex. If the algorithm is not resilient to the ordering, it must be aware of the number of messages to propagate per iteration. Note that a fourth category, where the algorithm is not ordering resilient, but the number of messages is not known, does not apply here, as ordering can only be guaranteed if it is certain that there is at least one message that will be received by the vertex per iteration, in which case the number would be known (one).

These properties are useful in determining how to apply KLA to a given algorithm and in understanding the behavior of the algorithm as k is varied. A large set of parallel graph algorithms can be broadly categorized into three categories based on the two properties. All algorithms in a category can be expressed in KLA in a similar manner and will exhibit similar behavior as k is varied. The categories, presented in decreasing order of penalty due to asynchrony (Table 1), are:

- **Type-I**: The number of incoming messages required to propagate a vertex is not known *a priori*. This type of algorithm optimistically propagates a vertex and processes every incoming message, forwarding the results to its outgoing edges, as the vertex is unaware of the total number of messages it may receive. If a different message is received by the vertex at a later time, it is also processed and could generate a new result, which may lead to re-doing some work for subsequent neighboring vertices. Algorithms in this category obtain the most benefit from KLA. Examples of this category include BFS, SSSP, BFS-based (betweenness centrality, connected components, etc.), minimum-spanning tree (GHS and Boruvka), and preflow-push maximum flow.
- **Type-II**: The exact number of incoming messages required to trigger propagation for each vertex is known *a priori*. Such algorithms avoid redoing work, as each vertex serves as a synchronization point and does not propagate until the required number of messages have been received.

  This category may be further subdivided into two types:

  - **Ordered-Type-II**: The output depends on the inter-level ordering of incoming messages. Therefore, messages may need to be buffered if they arrive out-of-order, and then processed once the required number of messages have been received. In such cases, greater asynchrony may lead to a greater occurrence of out-of-order messages, and thus may require greater buffering. Examples include PageRank, random walks, pointer jumping, graph coloring, and particle-swarm optimization.
  - **Unordered-Type-II**: This category includes algorithms where the ordering of incoming messages is not important, and it is sufficient to have received the required number of incoming messages in any order. Examples include k-core and topological sort.

In the next section, we present a model to study the behavior of each category and use it to determine a suitable k for algorithms in that category for a given input graph.

4. DETERMINING k

The performance of KLA algorithms is dependent on the choice of a good value for k. However, it is non-trivial to determine the optimal value of k given an arbitrary input graph, machine and algorithm. Therefore, we present a technique to quickly determine an effective approximate value for k_{opt}. We also present an adaptive strategy that picks an appropriate k based on the current local topology of the graph as the algorithm progresses. This strategy is applicable in the general case. The interpolative method to approximate the value of k_{opt} is useful when executing the algorithm multiple times on the input. Our experimental results show these two techniques work well in practice.

4.1 Adaptive Determination of k

To help users obtain better performance from a single (or a few) run(s) of a KLA algorithm, or when they do not have sufficient information about the input graph, we use a simple adaptive strategy (Adaptive KLA) that dynamically varies k to process the graph. The strategy dynamically chooses the best value for the next iteration based on information from current and previous iterations.

The algorithm starts with $k = 1$ (or some user-defined start value). At the end of each KLA-SS, the value of k is doubled for the next level under the following conditions: 1) high out-degree vertices have not been discovered, 2) the penalty for asynchrony (e.g., wasted work for Type-I or buffering cost for Ordered Type-II) has not exceeded a threshold, and 3) the cost of processing each vertex in the current KLA-SS is equal to or less than the cost of processing each vertex for the previous KLA-SSs. If the thresholds for either out-degree, asynchrony penalty or per-vertex processing cost have been exceeded, the k value is halved. In addition, any high out-degree vertices found in a given KLA-SS are marked for processing in the next KLA-SS to reduce the penalty for processing them prematurely. Finally, if the asynchrony penalty or processing cost exceed a maximum threshold (we empirically found 20% to work well for the machines on which we tested), the value for k is capped, so the algorithm does not suffer from greater asynchronous penalties attempting higher values. The thresholds for asynchrony penalty and processing cost can be chosen depending on the machine, such that for a machine where the cost of communication is much higher than the cost of computation, the threshold can be increased, and vice-versa.

While this technique may not provide the best performance compared to knowing *a priori* the value of k_{opt} (see Sec. 4.2), it is inexpensive in practice. As can be observed from Figure 11, it still provides substantial performance benefits compared to the level-synchronous and asynchronous paradigms by converging to an improved k value, which may then be used for subsequent runs.

4.2 A Model for Approximating k_{opt}

In this section we describe a model for the KLA paradigm that enables us to obtain a good approximation of the value (k_{opt}) for k which results in the lowest execution time for the algorithm based on the input graph and machine. This approach may be used when performing multiple traversals on

Type	Message Count	Message Ordering	Examples
Type-I	Unknown	Not Required	BFS, SSSP, MST, Preflow-Push, CC
Ordered Type-II	Known	Required	PageRank, Pointer Jumping, Graph-Coloring
Unordered Type-II	Known	Not Required	k-core, Topological Sort

Table 1: Characteristics of the three categories of KLA algorithms

a graph to obtain better performance than Adaptive KLA. We use BFS (Type-I) on an undirected, connected graph as an illustration of this model, and show, later in this section, how it applies to Type-II algorithms.

Theoretical Model. Assume a level-synchronous BFS requires d iterations over the graph, processing active vertices in each iteration. As there is an enforced ordering guarantee, there is no wasted work in this paradigm. Therefore, for a graph with n vertices on p processors, the work done in parallel can be V_{max}, the maximum number of vertices to be processed by a processor (ideally, $V_{max} = \frac{n}{p}$). Assuming it takes time α to process each vertex of the graph on average and T_{sync} is the time required by a global-synchronization of p processors in the system, the total time is given by:

$$T^{BFS}_{Level-Sync} = \alpha \cdot V_{max} + d \cdot T_{sync} \tag{1}$$

On the other hand, an asynchronous BFS traversal needs only one global-synchronization for termination detection, but in the process may end up doing redundant work due to loss of ordering guarantees for messages. Assuming a function $\Psi(G)$ gives the amount of redundant work done for graph G (a method for approximating this will be discussed in detail later), the total work per processor now becomes $V_{max} + \Psi^p(G)$, where $\Psi^p(G)$ is the estimated amount of wasted work on a processor. Therefore, the total time for the asynchronous paradigm is given by:

$$T^{BFS}_{Async} = \alpha \left(V_{max} + \Psi^p(G)\right) + T_{sync} \tag{2}$$

Extending this to the KLA paradigm, the traversal performs $\lceil \frac{d}{k} \rceil$ KLA supersteps. Assuming the wasted work for a given k is given by the penalty-function $\Psi(G,k)$, where $\Psi(G,1) = 0$ and $\Psi(G,d) = \Psi(G)$, the time now becomes:

$$T^{BFS}_{KLA}(k) = \alpha \left(V_{max} + \Psi^p(G,k)\right) + \lceil \frac{d}{k} \rceil \cdot T_{sync} \tag{3}$$

The product αV_{max} represents the time taken by the algorithm to complete all useful work in parallel, and can therefore be replaced by T_{Work}:

$$T^{BFS}_{KLA}(k) = T_{Work} + \alpha \cdot \Psi^p(G,k) + \lceil \frac{d}{k} \rceil \cdot T_{sync} \tag{4}$$

When $k = 1$, KLA behaves the same as the level-synchronous paradigm, and for $k = d$, it behaves like the asynchronous paradigm. In the following section, we provide some insight on what causes wasted work.

Modeling the Wasted Work, $\Psi(G,k)$. Wasted work is caused by latency, where a message taking a shorter path arrives after a message through a longer path. This can be caused by two major factors: machine dependent (i.e., network latency) and graph dependent (i.e., distribution of the graph, processing order of adjacent edges of a vertex). As the exact determination of these factors is non-deterministic, we provide a model to approximate the wasted work ($\Psi_{est}(G,k)$) for a given graph in this section.

We classify the edges of an input graph into two categories: E_T, which are the edges in the output BFS tree connecting parents to their children in the BFS, and $E_{NT} = E - E_T$, which are the other edges in the graph. There exist edges in E_{NT} then, that may cause wasted work, because, if these edges are traversed before the corresponding tree edges, the target vertex and its children will need to redo the traversal with shorter paths.

We can further sub-divide E_{NT} into E_{NT_C} and E_{NT_I}. E_{NT_C} are those E_{NT} edges that cross a KLA superstep, i.e., their source vertex is in a different KLA superstep than their target vertex. E_{NT_I} are the subset of remaining E_{NT} edges that have both their source and target inside the same KLA superstep, and therefore are inside an asynchronous execution section. As edges in E_{NT_C} cross a KLA superstep, they cross a global synchronization point. Therefore, they cannot contribute to wasted work, as their traversals will reach the target vertex at the same time as the correct traversal (from the actual BFS parent with the shortest distance-from-source), as guaranteed by the global synchronization. Therefore, we can establish an estimate of the potential wasted work as the cardinality of the set E_{NT_I}:

$$\Psi_{est}(G,k) \approxeq E_{NT\_I}(G,k) \tag{5}$$

We know the number of tree edges (E_T) equals $V - 1$. Therefore, $E_{NT} = E - (V - 1)$. Using this, we can approximate the expected number of non-tree edges inside a KLA-SS as:

$$E^{Expected}_{NT\_I}(G,k) = \frac{E - (V-1)}{d} \times (k-1) \tag{6}$$

From equations (5) and (6) we get:

$$\Psi_{est}(G,k) \approx \frac{E - (V-1)}{d} \times (k-1) \tag{7}$$

This supports our intuition. Small-diameter graphs with large out-degrees have $E >> V$ and $d = O(1)$. Therefore, even a slight increase in k may lead to large amounts of wasted work, which would negate the benefits gained from removing a global synchronization. This implies that such graphs will fair better with smaller k values ($k = 1$). Long-diameter graphs with low out-degrees have $E \approx V$ and $d = O(V)$. This implies that such graphs work best with $k = d$, as the wasted work is negligible. Other graphs such as road networks, geometric meshes, etc. tend to fall in-between, and therefore benefit the most with $1 < k < d$.

Furthermore, small-diameter graphs have a more even distribution of work, as most processors tend to be working in each superstep. Long-diameter graphs exhibit a work-load imbalance due to their low work per superstep. This imbalance can be alleviated by increasing the asynchrony for long-diameter graphs.

Ordered Type-II Algorithms. Type-II algorithms can be modeled in much the same way as Type-I algorithms, apart from the penalty paid for asynchrony. Ordered Type-II algorithms may incur a penalty for buffering messages

that arrive out-of-order as the asynchrony is increased. A vertex will have to buffer messages if at least one path that leads to it executes faster than the other paths leading to the same vertex. However, the amount of buffering is bounded to a maximum of k-levels. Therefore, the effect of buffering may be approximated using a linear penalty-function (see Eq. 7): $\Psi_{est.}(G,k) = m \times k + b$, where m and b are input-graph and machine dependent constants. Assume 1) the algorithm requires d iterations over the graph, 2) the time taken by the algorithm to complete all work in parallel is T_{Work}, 3) the cost of buffering is given by β (constant for a given system), and 4) T_{sync} is time required for global-synchronization of p processors in the system. The total execution time for ordered Type-II algorithms is given by:

$$T_{KLA}^{Ord-TII}(k) = T_{Work} + \beta \cdot \Psi_{est}(G,k) + \lceil \frac{d}{k} \rceil \cdot T_{sync} \quad (8)$$

Based on this formula, the constants m and b can be computed by running the algorithm twice with two different k values. This model is useful if the algorithm will be run multiple times iteratively, or for algorithms that iteratively perform the same computation (e.g. PageRank).

Unordered Type-II Algorithms. As there is no penalty for increasing asynchrony in unordered Type-II algorithms, we should ideally observe the best performance with maximum asynchrony. However, for certain graphs with large out-degrees, a completely asynchronous execution may cause network congestion due to a large number of messages in the system. In such cases, the level-synchronous execution should perform best. We observed this effect in social networks and other small-world scale-free graphs. Therefore, unordered Type-II algorithms could be run with $k = 1$ for such high out-degree graphs and asynchronously for other graphs, or using adaptive KLA, which can change k based on out-degree.

Finding k_{opt}: Estimating the Penalty-Function (Ψ_{est}) Determining Ψ exactly is expensive and only gives an upper-bound to the penalty paid for asynchrony. However, it can be closely approximated using a linear model (see Eq. 7). Assuming m gives an estimate of how fast the penalty increases with k, and b is the offset of the line: $\Psi_{est.}(G,k) = m \times k + b$ (both m and b depend on input-graph and machine). Two executions of the algorithm are needed to interpolate Ψ, which will provide the approximate curve of the running-time of the algorithm for varying values of k. The curve's minima can then be used to predict k_{opt}. This approach is mainly useful for application performing multiple traversals, such as in Brandes' betweenness centrality [6]. To find the equation for Ψ_{est}, we run the algorithm with $k = 1$, from which we can obtain the number of iterations (d) and the amortized cost (α) of processing a vertex. Running the algorithm a second time with $k = d$, which was found from the first execution, we can compute the penalty for a completely asynchronous execution of the algorithm ($\Psi_{est}(G,d)$), giving us the slope of the line for Ψ_{est}.

$\Psi_{est}(G,d)$ can be calculated for Type-I algorithms thus: 1) track the global number of visits to the input graph's vertices, 2) subtract from it the actual number of vertices in the graph, 3) divide by the number of processors to get the average wasted work per processor. For ordered Type-II algorithms, the number of visits that need to be buffered can be tracked and averaged across all processors. For unordered Type-II algorithms, the slope is zero.

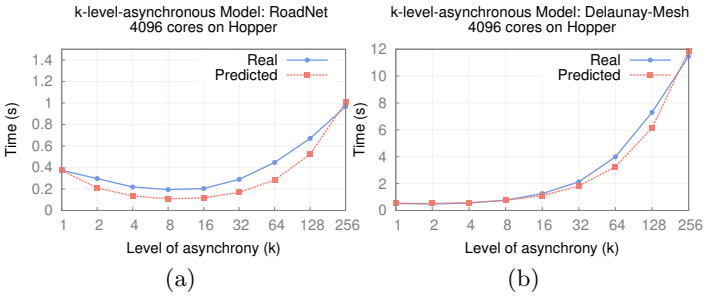


Figure 6: KLA Model: Predicted vs. Actual Execution Times on HOPPER for a Type-I algorithm (BFS).

Using these constants in the appropriate equation (Eq. 4 for Type-I algorithms and Eq. 8 for Type-II algorithms), we can obtain a curve approximating the performance of the algorithm for the given input graph, for different values of k. The k_{opt} is the value for which the curve reaches its minima. We ran KLA on multiple graphs and compared the predicted vs. real k_{opt}, and found the model to approximate the trend well enough to suggest a reasonable k_{opt} value. Figure 6 shows two such results comparing the predicted times given by the KLA model on two different input graphs, to the actual execution times for varying k-values. This model was used to predict k_{opt} for experiments in Sections 5.4 and 5.6.

5. EXPERIMENTS

In this section, we demonstrate the need for a single unifying paradigm with the ability to parametrically control asynchrony. We evaluate the performance of KLA as compared with traditional level synchronous and asynchronous paradigms for a variety of algorithms on various real-world and synthetic graphs.

5.1 Experimental Setup and Input Graphs

Experimental studies were performed on HOPPER – a Cray XE6 machine at the National Energy Research Scientific Computing Center (NERSC) with 153,216 total cores, of which 98,304 cores were available to us. Results reported were averaged over multiple runs to obtain a suitable confidence interval.

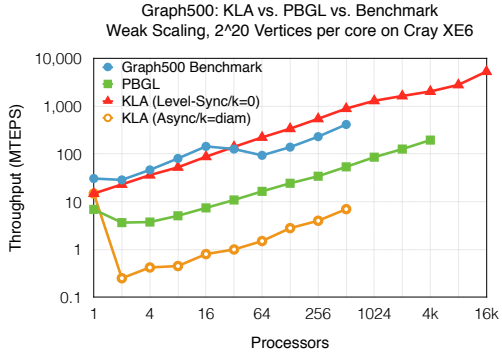
KLA was evaluated on representative graphs from different domains, ranging in size from a few million to tens of billions of vertices and edges (Table 2). These include the Graph 500 benchmark input [1], Google and Twitter web-graphs, various road-networks (TX, PA, US, EU and Synthetic) [2, 3], and geometric graphs (Delaunay meshes, geometric meshes and torii) [2, 3]. Input graphs use the default distribution provided by the input files.

5.2 SGL Overview

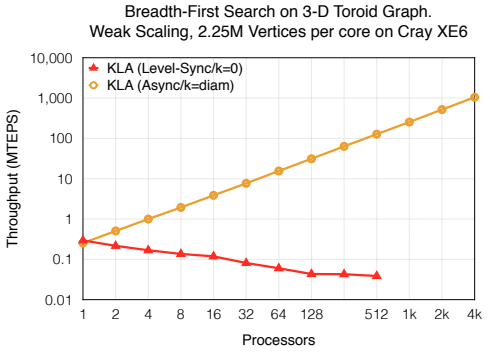
We implemented the KLA paradigm and some important graph algorithms in the STAPL Graph Library (SGL) [14] to evaluate its performance. SGL consists of a generic parallel graph container (`pGraph`), graph `pViews` [9], and a collection of parallel graph algorithms to allow users to easily process graphs at scale. The `pGraph` container is built using the `pContainer` framework (PCF) [29] provided by the Standard Template Adaptive Parallel Library (STAPL). STAPL [10] is a framework for parallel C++ code develop-

Name	Description	V, E	Source
G500-N	Graph500 Benchmark Input	2^N, 16×2^N	Graph 500 Benchmark
Google	Google Web-Graph	916K, 5.1M	SNAP
Twitter	Twitter Web-Graph	62M, 1.2B	SNAP
RoadNet-TX,PA	Road-networks (TX, PA)	(1.4M, 3.8M), (1M, 3M)	SNAP
RoadNet-US,EU	Road-networks (US, EU)	(24M, 58M), (50M, 54M)	Dimacs Challenge-9
RoadNet-Synthetic	Synthetic Road-network	9.63B, 10.2B	Generated
Delaunay	Delaunay Triangulation Graph	16M, 50M	Dimacs Challenge-10
RGG	Geometric Graph	16M, 132M	Dimacs Challenge-10
Torus	Toroid Mesh	$2.25M \times p$, $9M \times p$	Generated

Table 2: Characteristics of input graphs.



(a)



(b)

Figure 7: Scalability (Throughput) of BFS on (a) Graph500 benchmark (max. |V|=16B, |E|=256B) and (b) torus (max. |V|=9.2B, |E|=37B).

ment. STAPL's core is a library of components implementing parallel algorithms (`pAlgorithms`) and distributed data structures (`pContainers`).

5.3 Level-Synchronous vs. Asynchronous BFS

To demonstrate the performance differential between level-synchronous and asynchronous paradigms, we compared our implementation of an async BFS to its level-sync variant on a torus graph and the Graph 500 Benchmark graph (shown in Figure 7). These represent two extreme cases for long and short diameter graphs. We use $k = 1$ to emulate level-synchronous behavior and $k = n$ to emulate asynchronous behavior for KLA BFS. In both cases, the algorithm remains the same and only the value of k need vary to suit the input.

The Graph 500 input graph has a short diameter (<16) and vertices with very high out-degrees. For this graph, async BFS performs poorly due to the large number of messages flooding the system and the large amount of wasted work for revisited vertices. The level-sync BFS performs well in this case due to the input graph's low diameter, which implies fewer global synchronization. The torus graph, on the other hand, represents a worst-case scenario for parallel BFS scalability – the algorithm is essentially serialized due to the topology of the torus, which limits the available parallelism. In this scenario, we observed that async BFS performed much better than level-sync BFS, due to the absence of synchronization-points (Figure 7 (b)). These experiments demonstrate that the topology of the graph can significantly affect the performance characteristics of algorithms, and pairing the correct paradigm to the topology can result in a significant performance benefit.

5.4 Evaluation of KLA BFS

In this section we evaluate the performance benefit and scalability of KLA BFS on different types of real-world and synthetic graphs. We observed that for certain graphs, a value of $k > 1$ provides the best performance.

Graph Structure and k_{opt}. To investigate the correlation between the structure of a graph and the choice of k for the KLA algorithm, we ran the KLA BFS on a graph while deforming its shape to vary the diameter from very large (circular chain, $d \approx 128k$) to very small (random graph, $d \approx 8$). We obtained graphs with different diameters by allowing any given vertex to randomly select and connect only to its $\pm m$-closest neighboring vertices, varying m from 2 (circular chain) to n (random network). This is similar to the approach described by Watts and Strogatz [32]. We ran KLA BFS on each deformation for varying values of k, and observed which k was best suited for the given diameter. Figure 8 shows the value of k that gives the best performance for input graphs of varying diameter. This indicates that while a value of $k = 1$ better suits small-diameter graphs, there are also classes of graphs with intermediate diameters for which k_{opt} is between 1 and d. Road network graphs, analyzed in the next section, are one such class of graphs, along with scientific meshes. It is for such graphs that KLA provides improved performance, while maintaining the performance for other inputs and providing a single unifying algorithmic interface.

Road Network Graphs. KLA BFS can be used to accelerate finding routes on a road-map and navigation, as it is well suited to road network graphs due to their relatively long diameters. We obtained road networks for the

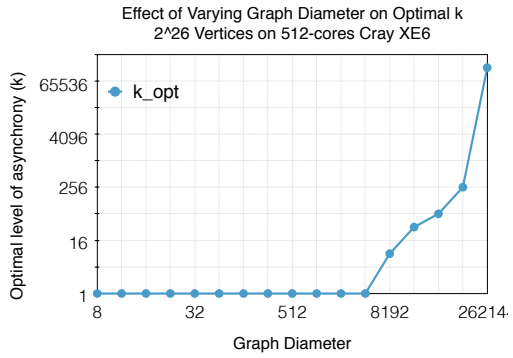


Figure 8: Modeling KLA: Study of effects of graph diameter on the optimal k parameter.

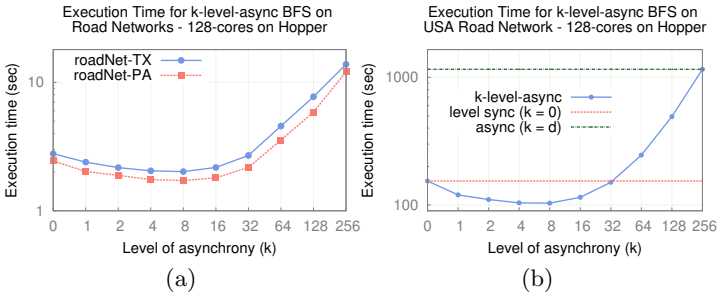


Figure 9: Performance of KLA BFS on road networks for (a) TX, PA and (b) USA on Hopper.

state of Texas and Pennsylvania (Figure 9(a)), and the entire U.S.A. (Figure 9(b)). For each of these graphs, the level-synchronous variant, with $k = 1$ proved expensive due to high number of synchronizations. However, owing to the amount of re-done work and the number of messages generated, the purely asynchronous version was also slow. In fact, the best performance was obtained with $1 < k < d$, where the computation was ≈27-33% faster than the fastest traditional paradigm for all three inputs.

Additionally, we generated a synthetic road network by stitching together multiple copies of the European road network, which provided us an input with 9.63 billion vertices and 10.2 billion edges. This input was large enough to allow for a meaningful strong-scaling study to 10^5 cores. Figure 10 compares the scalability of KLA BFS with level-synchronous BFS on this road network. The level-synchronous BFS scales to 32,768 cores, but not beyond. The KLA BFS is able to scale better until 98,304 cores (the maximum available cores). It yields faster running times due to better balancing synchronization costs, which become increasingly expensive for large core counts, with wasted work. This experiment demonstrates that traditional paradigms do not always scale to a large number of cores, where KLA can improve scalability and performance significantly.

5.5 Evaluation of Adaptive KLA BFS

For many applications, the optimal value of k may not be known for a particular input graph. Our Adaptive KLA technique provides improved execution times adaptively based on the graph's topology when users do not have sufficient information about the input graph.

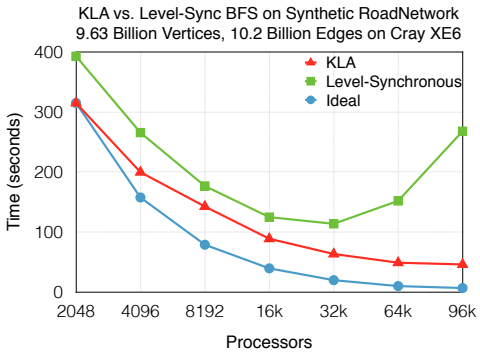


Figure 10: Performance of KLA vs. Level-Sync BFS on synthetic road network, 98,304 cores on Hopper.

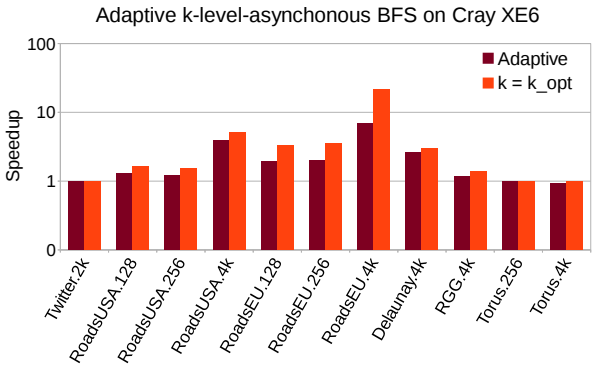


Figure 11: KLA BFS speedup with k_{opt} and Adaptive KLA on various graphs on Hopper.

We ran the Adaptive KLA BFS on input graphs from various domains, including road networks (Europe, USA), web graphs (Twitter) and geometric graphs (Delaunay, Random Geometric Graphs (RGG), Torus). We compared the speedup obtained using KLA BFS with k_{opt} to the Adaptive KLA algorithm for each graph. We also ran the level synchronous and asynchronous BFS on these graphs, and used the faster version's running-time as a baseline for calculating the speedup of both KLA BFS and Adaptive KLA BFS. Figure 11 shows the speedup obtained for various graphs over the best traditional paradigm. We observe that KLA with optimal k (k_{opt}) can provide improved performance over traditional paradigms for some input graphs. Due to the small girth of these graphs, not all processors are active in each iteration (this is especially true for higher processor counts) and those processors end up waiting at the global synchronizations in the level-synchronous approach. In contrast, the asynchronous approach may re-visit many vertices, increasing the run-time. KLA allows the traversal to proceed enough so as to recruit more processors in the iteration and yet controls the depth of asynchrony to avoid too many redundant visits, allowing for better scalability and performance. We also observe that Adaptive KLA can improve the performance, although the overhead of adaptivity (as it needs a few iterations to converge) and monitoring the iterations (for wasted work, etc.) keeps it from achieving the same level of performance as KLA with k_{opt}.

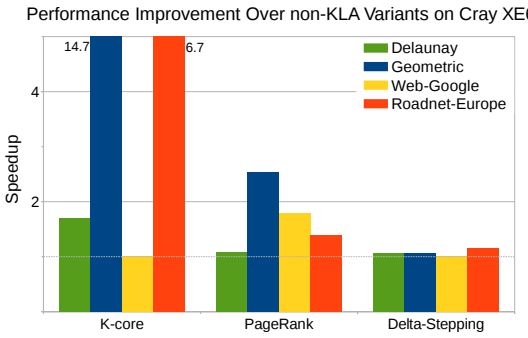


Figure 12: Evaluation of k-core PageRank and SSSP on 16,384 cores on HOPPER.

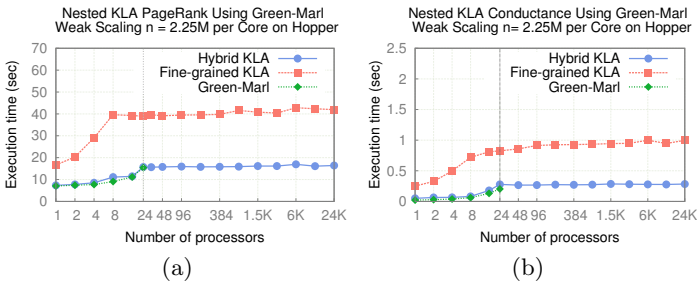


Figure 13: Nested KLA with Green-Marl for (a) PageRank and (b) conductance, up to 24,000 cores.

5.6 Evaluation of Other Types of Algorithms

In this section, we evaluate the performance of three important graph mining and graph analytics algorithms: k-core, PageRank and SSSP. We also compare the Δ-stepping SSSP algorithm with its KLA variant. We run the three algorithms on a variety of input graphs from different domains (road networks, scientific and geometric meshes, and web-graphs) and present their speedup (relative to level-synchronous versions in case of k-core and PageRank, relative to non-KLA Δ-stepping for SSSP) in Figure 12.

k-core. For k-core (Figure 12), we expect the fully asynchronous setting for k to perform fastest. While this is the case for most inputs, graphs with high out-degrees can cause a large number of messages to flood the network, leading to network congestion. In such cases, using the level-sync setting ($k = 0$) provides the best performance, as is observed for the Google web-graph. The benefit provided by KLA for k-core is the ease of algorithm tunability to obtain the best performance for any given input at runtime.

PageRank. PageRank incurs a penalty of buffering for increased asynchrony. However, as the cost of buffering was much lower than the cost for communication and global synchronization on our system, PageRank also saw good performance gains from the KLA paradigm (Figure 12).

Δ-stepping. We also compare an implementation of Δ-stepping SSSP in KLA with non-KLA Δ-stepping SSSP in Figure 12. The results presented show the speedup of the KLA version over the non-KLA version for the best values of Δ and k. We observe from these experiments that, as expected, both variants of Δ-stepping exhibit similar performance for different input graphs, as the KLA paradigm is a generalization of the Δ-stepping algorithm (Section 2.2).

5.7 KLA with Other Graph Frameworks

KLA with Green-Marl. KLA algorithms may also be called recursively in nested sections. We describe such computations using a tuple of k values. Each value in the tuple represents the k for each algorithm being executed in that level of nesting. The arity of the tuple denotes the number of levels of nesting in the algorithm. If the algorithm in the nested section is sequential or non-KLA, the k value for it is ignored ($k = x$). For example, using a shared-memory library for intra-node computation and KLA for inter-node computation, the tuple would be $\langle x, k_1 \rangle$. Figure 13 demonstrates the benefit of using nested-parallel execution in the nested KLA paradigm. It takes advantage of an optimized shared-memory library (Green-Marl) for processing partitions of the graph on a shared-memory node and uses KLA version of the algorithm on the upper-level to extend it across distributed nodes in the machine.

In our experiments, we ran two different algorithms – PageRank and cut-conductance – on a mesh graph using the Green-Marl [16] implementation within the node and KLA off-node, and compared them with their single-level KLA implementations in SGL.

KLA in Galois. In addition to the implementation provided for SGL, we illustrate the portability of the KLA technique by applying it within the Galois [15] parallel framework. Galois' execution model relies on shared-memory worklists that determine task scheduling of algorithms. A suite of worklists are provided by the library, including a bulk-synchronous worklist that contains two queues and processes them one after another, and an asynchronous worklist that contains several queues that may be processed in any order, but are prioritized by some metric (e.g. vertex-level).

We modified Galois' bulk synchronous worklist to process vertices in the same KLA-SS asynchronously and to only synchronize after k levels of asynchrony, effectively creating a KLA worklist for Galois. Figure 14 shows the results of executing KLA BFS in Galois for various input graphs. As with KLA in SGL, we see improvements for input graphs with long diameters, whereas short diameter graphs perform best with level-synchronous execution. In certain cases, the KLA worklist is able to achieve more than 90% speedup over the best of the level-synchronous and asynchronous worklists.

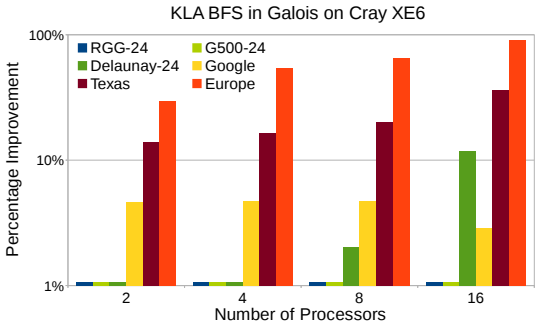


Figure 14: Speedup of KLA BFS in Galois on various graphs relative to the fastest version of Galois (bulk synchronous or asynchronous) on HOPPER.

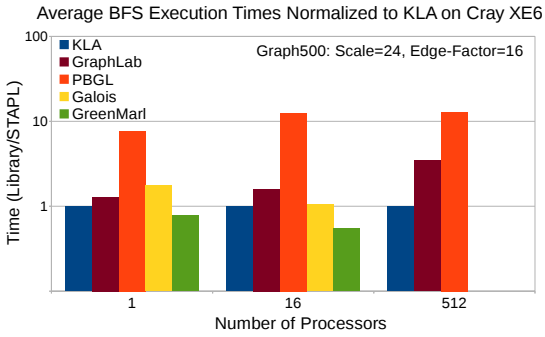


Figure 15: Execution times of Graph500 on various graph libraries normalized to SGL KLA on HOPPER.

6. RELATED WORK

Parallel graph algorithms have long been a subject of research in literature [18, 12]. Our work presents a *general, implementation-independent paradigm* that can provide substantial performance improvements for multiple important graph algorithms. The paradigm itself is applicable to any graph framework supporting asynchronous execution, and while we focus on vertex-centric algorithms, it may also be extended to edge-centric frameworks, as well as expressed in graph-specific languages (graph DSLs). In this work, we mainly focus on implementations using the STAPL Graph Library, however, Section 5.7 shows applicability of KLA to Galois and Green-Marl.

This section reviews some of the relevant parallel graph-processing frameworks. We show a comparison of running times of BFS on the standard Graph 500 benchmark for these libraries with our KLA implementation in Figure 15.

There has been much research in using a level-synchronous approach to express graph algorithms. Most popular graph libraries such as Google's Pregel [21], Parallel Boost Graph Library (PBGL) [13, 11], Multi-Threaded Graph Library (MTGL) [5], along with most stand-alone implementations of parallel graph algorithms (e.g. Graph500 reference implementation [1]), use the level-synchronous paradigm to express and implement parallel algorithms. This paradigm makes it easier to reason about and debug graph algorithms in parallel due to the guarantees it provides, and is straightforward to implement. A detailed comparison of SGL with these libraries was shown in [14]. GRACE [31] and Galois [15] are other shared-memory graph libraries. GRACE allows asynchronous execution while abstracting users from the execution policies. Galois uses optimistic set iterators and operator scheduling to execute algorithms. However, this results in Galois requiring users to handle low-level concurrency issues such as data sharing and deadlock avoidance.

The asynchronous paradigm has also been applied to various graph algorithms [12, 24]. However, they may not be as straightforward to express and reason about as their level-synchronous counterparts. Recently, there have been a few graph processing frameworks that allow asynchronous execution of the algorithms [14, 20, 28, 31]. GraphLab/PowerGraph [20] is one such distributed-memory framework that allows users to choose between synchronous and asynchronous execution at runtime, but it does not support parametric control of asynchrony. It also exposes 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. [22] presents Δ-stepping SSSP, which is discussed in Section 2.2 in detail.

There has even been work on creating domain-specific languages (DSLs) for graph processing. Green-Marl [16] is one such DSL for graph analysis which allows users to express graph-based computations naturally, abstracting parallelism details. The Green-Marl compiler is able to generate code for specific target languages and libraries, which can produce algorithm-specific code with less overhead than the fully generic code in STAPL. It provides an implementation for shared-memory systems using OpenMP. Such a DSL could be extended to generate code for KLA and SGL, allowing users of Green-Marl access to distributed-memory systems, as shown in Section 5.7. Elixir [25] is another DSL which produces Galois code and is designed for concurrent graph processing for shared-memory systems. It also supports loop-unrolling for certain graph algorithms at compile-time, which can reduce global synchronizations in the output code. However, being a compile-time optimization, the algorithm would need to be recompiled for each input with a suitable unroll-factor, which is not a realistic solution.

While it is well known that there exist two ways (asynchronous and level-synchronous) of expressing parallel graph algorithms, the effects of parametrically controlling asynchrony as a general paradigm has not been studied.

7. CONCLUSION

In this paper we have presented a novel approach to designing and implementing parallel graph algorithms — the k-level async (KLA) paradigm — that enables the level of asynchrony of parallel graph algorithms to be parametically varied from none (level-synchronous) to full (asynchronous). We provided guidance on how to express graph algorithms in the KLA paradigm and provided techniques for determining k, the number of asynchronous steps allowed between global synchronizations. Our experimental results show KLA implementation of graph algorithms such as BFS, PageRank and k-core provide good scalability on up to 98,000 cores and improvements of 10x or more over level-synchronous and asynchronous versions of these algorithms.

8. ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers. This research is supported in part by NSF awards CNS-0551685, CCF-0833199, CCF-0830753, IIS-0916053, IIS-0917266, EFRI-1240483, RI-1217991, by NIH NCI R25 CA090301-11, by DOE awards DE-AC02-06CH11357, B575363, by Samsung, Chevron, IBM, Intel, Oracle/Sun and by Award KUS-C1-016-04, made by King Abdullah University of Science and Technology (KAUST). This material is based upon work supported by the U.S. Department of Energy, National Nuclear Security Administration, under Award Number(s) DE-NA0002376, and used resources of the National Energy Research Scientific Computing Center, which is supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

9. REFERENCES

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

[2] Stanford Large Network Dataset Collection. http://snap.stanford.edu/data/index.html, 2013.

[3] 9^{th} DIMACS Implementation Challenge. http://www.dis.uniroma1.it/challenge9/, 2013.

[4] 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.

[5] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny. Software and algorithms for graph queries on multithreaded architectures. In *Intl. Parallel and Distributed Processing Symp.*, 0:495, 2007.

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

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

[8] A. Buluç and K. Madduri. Parallel breadth-first search on distributed memory systems. In *Proc. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '11, pp. 1–12, New York, NY, USA, 2011.

[9] A. Buss, A. Fidel, Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. The STAPL pView. In *Intl. Workshop on Languages and Compilers for Parallel Computing (LCPC), in Lecture Notes in Computer Science (LNCS)*, Houston, TX, USA, 2010.

[10] 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. Annual Haifa Experimental Systems Conference (SYSTOR)*, pp. 1–10, New York, NY, USA, 2010.

[11] N. Edmonds, J. Willcock, and A. Lumsdaine. Expressing graph algorithms using generalized active messages. In *Proc. of Symp. on Principles and Practice of Parallel Programming*, PPoPP '13, pp. 289–290, New York, NY, USA, 2013.

[12] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. In *Trans. Program. Lang. Syst.*, pp. 66–77, 1983.

[13] D. Gregor and A. Lumsdaine. The Parallel BGL: A generic library for distributed graph computations. In *Parallel Object-Oriented Scientific Computing*, POOSC, 2005.

[14] Harshvardhan, A. Fidel, N. M. Amato, and L. Rauchwerger. The STAPL Parallel Graph Library. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science, pp. 46–60. Springer Berlin Heidelberg, 2012.

[15] M. A. Hassaan, M. Burtscher, and K. Pingali. Ordered and unordered algorithms for parallel breadth first search. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT '10, pp. 539–540, New York, NY, USA, 2010.

[16] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun. Green-Marl: A DSL for easy and efficient graph analysis. In *Proc. of the Intl. Conf. on Architectural Support for Prog. Languages and Operating Syst.*, ASPLOS'12, pp. 349–362, New York, NY, USA, 2012.

[17] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration for multi-core cpu and gpu. In *Proc. of the Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT '11, pp. 78–88.

[18] J. JàJà. *An Introduction Parallel Algorithms*. Addison–Wesley, Reading, Massachusetts, 1992.

[19] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. of the Symp. on Parallelism in Algorithms and Architectures*, SPAA '10, pp. 303–314, New York, NY, USA, 2010.

[20] 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. *Proc. of the VLDB Endowment*, pp. 716–727, 2012.

[21] 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. of the Intl. Conf. on Management of data*, SIGMOD '10, pp. 135–146, New York, NY, USA, 2010.

[22] U. Meyer and P. Sanders. Delta-stepping : A parallel single source shortest path algorithm. In *ESA '98: Proc. of the European Symp. on Algorithms*, pp. 393–404. Springer-Verlag, 1998.

[23] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. 1998.

[24] R. Pearce, M. Gokhale, and N. M. Amato. Multithreaded asynchronous graph traversal for in-memory and semi-external memory. In *Proc. of Intl. Conf. for High Performance Computing, Networking, Storage and Analysis*, SC '10, pp. 1–11, Washington, DC, USA, 2010.

[25] D. Prountzos, R. Manevich, and K. Pingali. Elixir: A system for synthesizing concurrent graph programs. In *Proc. of the Intl. Conf. on Object Oriented Program. Syst. Languages and Applications*, OOPSLA '12, pp. 375–394, New York, NY, USA, 2012.

[26] M. J. Quinn and N. Deo. Parallel graph algorithms. In *ACM Computing Surveys (CSUR)*, pp. 319–348, 1984.

[27] J. H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufmann, San Mateo, CA, 1993.

[28] P. Stutz, A. Bernstein, and W. Cohen. Signal/collect: Graph algorithms for the (semantic) web. In *The Semantic Web–ISWC '10*, pp. 764–780. Springer, 2010.

[29] 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. of Symp. on Principles and Practice of Parallel Programming*, PPoPP, pp. 235–246, San Antonio, TX, USA, 2011.

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

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

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