Graph Colouring as a Challenge Problem for Dynamic Graph Processing on Distributed Systems

Scott Sallinen$^{1,2}$, Keita Iwabuchi$^{1,3}$, Suraj Poudel$^{1,4}$, Maya Gokhale$^1$, Matei Ripeanu$^2$, Roger Pearce$^1$
$^1$Lawrence Livermore National Laboratory, $^2$University of British Columbia, $^3$Tokyo Institute of Technology, $^4$University of Alabama at Birmingham

Abstract—An unprecedented growth in data generation is taking place. Data about larger dynamic systems is being accumulated, capturing finer granularity events, and thus processing requirements are increasingly approaching real-time. To keep up, data-analytics pipelines need to be viable at massive scale, and switch away from static, offline scenarios to support fully online analysis of dynamic systems.

This paper uses a challenge problem, graph colouring, to explore massive-scale analytics for dynamic graph processing. We present an event-based infrastructure, and a novel, online, distributed graph colouring algorithm. Our implementation for colouring static graphs, used as a performance baseline, is up to an order of magnitude faster than previous results and handles massive graphs with over 257 billion edges. Our framework supports dynamic graph colouring with performance at large scale better than GraphLab’s static analysis. Our experience indicates that online solutions are feasible, and can be more efficient than those based on snapshotting.

I. INTRODUCTION

Past research to improve support for sophisticated graph analytics has led to remarkable advances in processing speed, ability to scale, and support for fast solution prototyping. Much of this work has used HPC systems [1][2][3] and is showcased by the bi-annual Graph500 competition [4]. However, as these approaches have been designed for static graphs, directly using them for dynamic graphs is not possible without using coarse, and problematically inefficient, static snapshots.

We explore support for online dynamic graph processing through an event-centric infrastructure: when the graph structure or vertex/edge attributes change, the infrastructure triggers an algorithmic event that allows user-defined callbacks to perform the necessary application updates. To support this exploration, we have extended HavoqGT [5], an open-source graph analytics framework that provides infrastructure to develop asynchronous vertex-centric graph algorithms, to additionally support dynamic graphs: including ingesting, routing, and managing the algorithmic events generated.

As analytics for dynamic graphs is a relatively unexplored area at both the infrastructure- and algorithmic-levels, we have chosen an approach based on a challenge-problem to drive this investigation. Our choice of graph-colouring as the challenge problem is motivated by its importance and algorithmic properties. Problems in multiple areas that are conceptually an allocation strategy (e.g., scheduling, independence testing, resource allocation) can be represented as graphs: when edges represent scheduling conflicts, a colouring becomes a solution to the allocation problem. Graph colouring itself, that is, using the minimal number of colours to colour each vertex so that no two adjacent vertices have the same colour, is NP-Hard [6]. There are, however, many polynomial-time greedy heuristics that achieve a ‘good’ colouring: the number of colours used is close to minimal, while the colouring is still correct (i.e., adjacent vertices have different colours). A modern domain that stresses the need for scale and dynamicity of graph colouring is machine learning. Sample independence is often used to build an effective training set, and graph colouring can be used to select only non-similar samples in the training set.

In situations where training samples are acquired dynamically, an online graph colouring solution is needed.

This paper explores support for heuristics for graph colouring on two key axes: (i) static and dynamic graphs, and (ii) scale. We use real-world data to provide novel insight into a fully on-line graph colouring solution for dynamic graphs. Importantly, unlike previous solutions, our solution is edge-centric: it allows adding and removing individual edges. We focus on a distributed setting and massive-scale graphs, using the well-established first-fit greedy colouring heuristic.

This paper makes the following key contributions:

- **Colouring for static graphs**: To create a performance baseline for our colouring solution of dynamic graphs, we first detail a solution to scale-up static graph colouring (Section II) and provide a detailed performance characterization (Section III). Our distributed processing engine can efficiently colour static graphs at an extreme scale: we test our algorithms on the largest existing public graph (WebGraph [7]). At over 257 billion edges, this is two orders of magnitude larger than previously seen in literature for graph colouring. For all graphs we colour, our approach significantly outperforms the implementation by GraphLab [8].

- **Online algorithmic framework**: We propose a novel framework to support online graph algorithms for dynamic graphs (Section IV). As vertex/edge attributes or the graph structure change (e.g., edges are inserted or deleted) the framework triggers algorithmic events allowing on-line algorithms to perform the necessary updates (as specified by the user). We demonstrate that this framework can efficiently support an online graph colouring heuristic (Section V). As there is relatively little past practical work in this direction [9], we hope that the availability of this framework will generate momentum: and create an environment where online graph algorithms...
• Novel online colouring heuristic for dynamic graphs: We propose and evaluate a novel on-line, edge-centric colouring heuristic for dynamic graphs (Section V). We evaluate this heuristic by comparing dynamic execution on static graphs (being up to 9.7x faster than GraphLab at high node count), as well as on a real dynamic graph, in section VI.

• Curated dataset for dynamic graphs: To test our algorithms at scale with real-world data, we have curated the largest existing dynamic (or temporal) graph data set from over a decade of the English Wikipedia corpus. In this data set, vertices are Wikipedia pages, and edges are hyperlinks with start and end time stamps corresponding to the edge creation and possible deletion events.

II. IMPLEMENTING STATIC GRAPH COLOURING

Many recent graph analytics frameworks have explored the design space of single node, shared memory systems [10][11][12][13]. Shared memory enables pull access: a vertex can directly explore its neighbours state. More concretely, if vertex A has an edge to vertex B, in a shared memory system the process controlling A can access an index that holds the global storage location of B and directly obtain its current state. In contrast, in a push-based access, vertex B must send its state to vertex A, a style useful in an environment with distinct address spaces. The rest of this section describes the push-based algorithm used, and our implementation on top of HavoqGT graph-processing framework.

A. The Heuristic Used for Static Graph Colouring

The starting point for our work is the greedy colouring heuristic presented by Jones et al. [14]. There are multiple reasons to start from this heuristic: (i) it is efficient, (ii) it allows an asynchronous push-based implementation that fits well a distributed memory platform, and (iii) it leads to a high quality solution (past work shows that it achieves near optimal colouring in many cases [15]).

The heuristic works as follows. Each vertex is assigned a priority. Then each vertex waits for higher priority neighbours to colour themselves and announce their colour, then colours itself based on received neighbours’ colour information. If two neighbouring vertices have the same priority, the vertex ID is used to break ties. The key advantage this strategy offers is that all uncoloured vertices that are ready to choose their colour (i.e., have maximum priority among their neighbours) can do so concurrently, without any synchronization. This property makes this heuristic ideally suited for a distributed memory platform.

The two stages of the algorithm are described below and their runtime properties are analyzed in Section III.

• Stage 1: Collect Priorities (Algorithms 1 and 2). Each vertex computes the number of neighbours with higher priority. This is generally done by having each vertex pass a message to each of its neighbours (once), announcing its globally unique priority. Each vertex checks through all incoming messages, and determines how many vertices have a higher priority than its own.

There are multiple ways to assign vertex priority, for example, Wright et al. [16][14] compare multiple parallel methods including the largest-degree-first (LDF) [17] (Algorithm 1), where the priority of a vertex is its own degree (hence, the largest degree vertices have the highest priority, and choose their colour first). Other strategies, such as a random priority assignment, are also feasible [16][14]. One strategy in particular has an added benefit: hashing the (unique) global vertex ID (Algorithm 2). Given a uniform hash function, this essentially produces a random ordering. On a distributed platform, however, this approach offers a key advantage: a vertex can compute the priorities of its neighbours directly (by hashing their IDs) without any message exchange. As Section III shows, using this approach drastically reduces the runtime for the first step, since no messages are sent, and priorities can be computed in parallel across all vertices.

• Stage 2: Colour (Algorithm 3) In this stage all vertices that have a count of zero (i.e., none of their neighbours has a higher priority) choose their colour. The colour choice at a vertex is determined by knowing the colour of higher priority neighbours that have already chosen their colour. Immediately after stage one, the first vertices, knowing that all their neighbours are uncoloured, choose the ‘first’ colour. These vertices then notify all of their neighbours with the chosen colour. On receipt of this message, a vertex will add the colour to its unusable list (represented as a bitmap that is the size of the vertex’s degree+1), as well as decrement their counter. If the counter reaches zero, the vertex ‘activates’ and can colour itself.

B. Implementation on Top of Asynchronous Visitor Queue

HavoqGT (Highly Asynchronous Visitor Queue Graph Toolkit) [5] provides a distributed framework that supports implementing graph algorithms using an asynchronous visitor abstraction. The framework targets parallel and distributed environments and large scale-free graphs, with optimizations for dynamic [18] and temporal [19] graphs. Pearce et al. [20] demonstrated that this abstraction and its implementation provide excellent scalability [21].

The visitor abstraction allows an algorithm designer to define vertex-centric procedures that execute on traversed vertices, and offers the ability to pass visitor state to other vertices [21]. The visitor procedures and the state to be defined in the visitor are summarized in Table I. All algorithm implementations presented here follow this abstraction. When an algorithm needs to traverse to another vertex, it dynamically creates a new visitor and pushes it into the visitor queue. When an algorithm begins, an initial set of visitors is pushed on the queue and the framework’s driver invokes the traversal. The asynchronous traversal completes when all visitors have

<table>
<thead>
<tr>
<th>TABLE I: HavoqGT Visitor Abstraction</th>
</tr>
</thead>
<tbody>
<tr>
<td>pre_visit</td>
</tr>
<tr>
<td>visit</td>
</tr>
<tr>
<td>vertex</td>
</tr>
<tr>
<td>performs a preliminary evaluation of</td>
</tr>
<tr>
<td>the state and returns true if the</td>
</tr>
<tr>
<td>visitation should proceed.</td>
</tr>
<tr>
<td>main visitor procedure</td>
</tr>
<tr>
<td>stored state representing the vertex</td>
</tr>
<tr>
<td>to be visited.</td>
</tr>
</tbody>
</table>
completed, which is determined by a distributed quiescence detection algorithm.

III. ANALYZING STATIC GRAPH COLOURING

Apart from establishing a baseline used later to evaluate the online solution, this section evaluates our static graph colouring heuristic and demonstrates its scaling properties (weak and strong-scaling experiments), compares to a well-known graph processing framework (GraphLab), and demonstrates ability to colour a massive-scale 257 Billion-edge graph, when stored in memory or using NVRAM.

A. Experimental Platform, Methodology, Workloads

Experimental platform: Our experimental platform is the Catalyst cluster at Lawrence Livermore National Laboratory. Catalyst is a 324 node experimental data-intensive platform that has dual 12-core Intel Xeon E5-2695v2 (2.4 GHz) processors, 128 GB of memory, and Intel 910 PCI-attached NAND Flash per node.

Experimental methodology: Plots present averages over 10 runs for runtime. The colour count is deterministic – due to this, the standard deviations on runtime are small, and not presented here to reduce clutter. For all HavoqGT executions, we discount all pre-processing; further, for GraphLab, we discount all pre-processing.

Workloads: For evaluation we use large real-world and synthetic graphs including largest existing real-world graphs available. These are presented in detail in Table II.

B. Scaling.

Figures 1 and 2 present runtime for each stage of the heuristic presented in the previous section (Collect and Colour stages) for weak and strong scaling experiments. These experiments also evaluate the impact of the method to assign vertex priority: the ‘Largest Degree First’ (LDF) and Hashing methods.

For this experiment we use synthetic Erdös-Rényi graphs (vertices have uniform degree distribution). Graph scale and the number of compute nodes used are indicated on X-axis as (scale,nodes) pairs in Figure 1. Graph scale is defined as 2^{scale} vertices, and we use an average degree of 16, for 2^{scale} edges, as edges are undirected. Scale 32 for example has 137 billion edges.

Figure 1 also plots the ratio of non-local edges: assuming an even distribution of edges across processes, on average the number of edges that a process owns is (E/P). Thus, the percentage of non-local edges is (P − 1)/P. Unsurprisingly, as scale increases, the ratio of non-local edges has a similar trend as processing time: as more edges are non-local, the processing becomes bounded by communication.

In Figure 2, strong scaling, we keep the size of the graph constant (Scale28) but vary the number of computing nodes. On Y-axis, the figure presents the total compute time (logarithmic scale), thus a diagonal line in the plot is an evidence of adequate strong scaling.

C. Understanding Colour Count.

Moving beyond an Erdös-Rényi graph, we generated synthetic RMAT graphs with varying degrees of properties. In Figure 3, we show four synthetic generated RMAT graphs, with increasing the power law nature of each graph. The requirement for more colours (leading to longer runtime as less parallelism is available) with an increasing power law nature is an expected property. When select few vertices become highly interconnected, priority based graph colouring slows down. One can imagine colouring a clique of size N; in any colouring method, the end result is a sequential colouring, using all N colours (as all members of the clique must be coloured a different colour). Thus, as we increase the size of

<table>
<thead>
<tr>
<th>Name</th>
<th>#Vertices</th>
<th>#Edges</th>
<th>OnDiskSpace</th>
</tr>
</thead>
<tbody>
<tr>
<td>Friendster [23]</td>
<td>65,608,366</td>
<td>3,612,134,270</td>
<td>61 GB</td>
</tr>
<tr>
<td>Twitter [23]</td>
<td>41,652,230</td>
<td>2,936,729,768</td>
<td>49 GB</td>
</tr>
<tr>
<td>WebGraph [25]</td>
<td>3,563,602,686</td>
<td>257,473,828,334</td>
<td>5.1 TB</td>
</tr>
<tr>
<td>RMAT(SCALE)</td>
<td>2^{SCALE}</td>
<td>2^{SCALE} * 32</td>
<td></td>
</tr>
<tr>
<td>e.g. RMAT32</td>
<td>4,294,967,296</td>
<td>137,438,953,472</td>
<td></td>
</tr>
</tbody>
</table>

TABLE II: Graphs used in experiments. RMAT graphs have a 16x edge factor. All graphs are made undirected.

Algorithm 1: Collect priority: LDF

```plaintext
1 bool PRE_VISIT() {
2    if (firstSelfVisit) {
3        return true; // VISIT once.
4    } else {
5        my_priority = graph.degree_of(this);
6        if ((caller_priority > my_priority) ||
7            && caller.ID > this.ID))
8            this.wait_count++;  
9            return false;  
10       }
11       }
12
13 void VISIT(graph, queue) {
14    for (all outgoing edges E) {
15        caller_priority = graph.degree_of(this);
16        queue.insert(E, caller_priority);
17        }
18    }
19}
```

Algorithm 2: Collect priority: Hash

```plaintext
1 void PRE_VISIT() {
2    my_priority = hash(this.ID);
3    for (all outgoing edges E) {
4        edge_priority = hash(E.ID);
5        if (edge_priority > my_priority ||
6            && E.ID > this.ID))
7            this.wait_count++;  
8            return false;  
9            }
10       }
11
12 void VISIT(graph, queue) {
13    for (all outgoing edges E) {
14        my_priority = hash(this.ID);
15        edge_priority = hash(E.ID);
16        if (firstSelfVisit && this.wait_count == 0) {
17            return true; // VISIT once.
18        } else if (this.wait_count == 0) {
19            return true; // Just became ready to colour.
20            }
21        }
22    }
23}
```

Algorithm 3: Colouring

```plaintext
1 bool PRE_VISIT() {
2    if (firstSelfVisit && this.wait_count == 0) {
3        return true;
4    } else if (this.wait_count == 0) {
5        return false; // Already coloured.
6    } else {
7        this.nbr_colours.append(caller_colour);
8        this.wait_count--;  
9        if (this.wait_count == 0)
10           return true;  
11       }
12       }
13
14 void VISIT(graph, queue) {
15    this.colour = findFirstUnused(this.nbr_colours);
16    for (all outgoing edges E) {
17        queue.insert(E, this.colour);
18    }
19}
```

```plaintext
10 runs for runtime. The colour count is deterministic – due to this, the standard deviations on runtime are small, and not presented here to reduce clutter. For all HavoqGT executions, we discount all pre-processing; further, for GraphLab, we discount all pre-processing.

Workloads: For evaluation we use large real-world and synthetic graphs including largest existing real-world graphs available. These are presented in detail in Table II.

B. Scaling.

Figures 1 and 2 present runtime for each stage of the heuristic presented in the previous section (Collect and Colour stages) for weak and strong scaling experiments. These experiments also evaluate the impact of the method to assign vertex priority: the ‘Largest Degree First’ (LDF) and Hashing methods.

For this experiment we use synthetic Erdös-Rényi graphs (vertices have uniform degree distribution). Graph scale and the number of compute nodes used are indicated on X-axis as (scale,nodes) pairs in Figure 1. Graph scale is defined as 2^{scale} vertices, and we use an average degree of 16, for 2^{scale} edges, as edges are undirected. Scale 32 for example has 137 billion edges.

Figure 1 also plots the ratio of non-local edges: assuming an even distribution of edges across processes, on average the number of edges that a process owns is (E/P). Thus, the percentage of non-local edges is (P − 1)/P. Unsurprisingly, as scale increases, the ratio of non-local edges has a similar trend as processing time: as more edges are non-local, the processing becomes bounded by communication.

In Figure 2, strong scaling, we keep the size of the graph constant (Scale28) but vary the number of computing nodes. On Y-axis, the figure presents the total compute time (logarithmic scale), thus a diagonal line in the plot is an evidence of adequate strong scaling.

C. Understanding Colour Count.

Moving beyond an Erdös-Rényi graph, we generated synthetic RMAT graphs with varying degrees of properties. In Figure 3, we show four synthetic generated RMAT graphs, with increasing the power law nature of each graph. The requirement for more colours (leading to longer runtime as less parallelism is available) with an increasing power law nature is an expected property. When select few vertices become highly interconnected, priority based graph colouring slows down. One can imagine colouring a clique of size N; in any colouring method, the end result is a sequential colouring, using all N colours (as all members of the clique must be coloured a different colour). Thus, as we increase the size of
Fig. 1: Weak Scaling Experiments using Erdős-Rényi graphs. The plot presents runtime for each stage of the algorithm for LDF and Hash priorities. X-axis labels represent \[\text{scale, nodes}\]. Also shown is the percentage of edges that are non local for a given node count. A flat line indicates perfect weak scaling.

Fig. 2: Strong Scaling Experiment using an Erdős-Rényi Scale 28 graph. Y-axis represents runtime (log-scale, unlike the weak scaling plot). X-axis represents number of nodes used. A straight diagonal line parallel with the grey line indicate perfect strong scaling.

D. Comparison to GraphLab Processing Framework

We also compared our baseline static processing performance to that GraphLab \[26\], one of the modern graph processing frameworks. We use the ‘simple colouring’ method: we were unable to compare their other methods as they crashed with larger graphs. Figure 3 plots runtime for GraphLab and our method (using both Hash and LDF vertex priority assignment schemes). The colour count for each solution was generally similar on a given graph, with GraphLab generally performing worse, followed closely by the Hash method, and LDF consistently offering the best solution (fewest colours). The LDF and Hash methods were both generally similar in performance, with the Hash method always being faster due to the ability to calculate priority without sending messages. GraphLab performed the worst in all cases: being 11.9x slower than our Hash method on the more uniform graph (even distribution of degrees), and up to 9.6x slower on the least uniform graph (skewed distribution of degrees).

E. Real World Graphs.

We also evaluated GraphLab on several real world graphs. In Figure 4 we show Twitter, Friendster, and SK2005, with properties as in table II. Similar to the synthetic graphs, GraphLab did not scale well with an increasing colour count - with the SK2005 dataset, which resulted in about 4511 colours, GraphLab actually negatively scaled with increasing node count. Also surprising, despite the datasets represented in src/dst pair text documents of sizes shown in table II GraphLab was not able to execute these graphs on fewer than 4 nodes, despite each node having 128GB of main memory. In HavoqGT, we can execute these small graphs in main memory without problem, and in fact HavoqGT can support even larger graphs on a single node by using NVRAM (discussed in the next section).

F. Processing a Massive Graph

Previous work \[27\][28][29][30][31] on distributed colouring has focused on small graphs, similar (or the same as) the graphs we presented in the previous subsections. Although optimizing at a small scale (such as for graphs that fit in accelerator memory \[32\]) is also important, we aim to support colouring of a much larger scale. To this end, we use a real world Web Data Commons Hyperlink Graph, (Page-Level) \[25\]. This graph has over 257 Billion edges, and is almost two orders of magnitude larger than any of the graphs presented in the previous studies (between 20,000x and 60x larger). We attempted to load the graph in GraphLab, and even with our hand-tweaking and modifying parts of the source code, we were unable to colour the graph (due to out of memory crashes), even trying to load on 300 nodes (totalling 38.4TB aggregate memory).

Figure 5 shows the runtime for colouring the Webgraph for multiple configurations (storing the graph in memory or on NVRAM). (The plot also includes as the right-most bars, the runtime for colouring the dynamic graph – adding edges one at the time – as discussed in Section VI). Note that below 32 nodes, the graph data (in memory efficient CSR format) no longer fits into memory. Despite this challenge, HavoqGT can process the graph from NVRAM instead. Below 4 nodes, the graph partitions no longer fit into a node’s NVRAM (this is a limitation of our hardware, not the framework).

Apart from demonstrating the ability to process massive scale graphs in reasonable time, there are a number of observations this plot supports: first, scaling tends to flatten out at larger node counts, likely due to the highly power law nature of the graph (the graph was coloured with 10,566 colours by both static methods). Second, is the impact of using NVRAM: for the Hash method, the Collect phase was up to 4s when the graph was stored in memory; however, when reading from NVRAM, the Collect phase took between 280s and 832s - a massive jump. As this phase does not need communication, it shows that for our system, IO is a significant overhead when processing with NVRAM.
Fig. 3: Run-time and number of colours for RMAT graphs with varying power-law nature (less-pronounced – left, to more-pronounced – right). The experiment uses 8 nodes and a Scale 27 graph. Collect and Colour phases are stacked. (GraphLab – labelled GL, does not give distinct information; Collect-Hash is not visible as it is too fast.) The number of colours used is presented in boxes. The time for the final (right-most) GraphLab run was 919s, about three times the size of the bar shown.

(a) Twitter. About 1120 colours.
(b) Friendster. About 130 colours.
(c) SK2005. About 4511 colours.

Fig. 4: Comparison to GraphLab. The plots present runtime for GraphLab and our solution using LDF and Hash vertex priority assignment for 1 to 64 nodes. Missing data-points in the plot indicate that GraphLab was unable to load the data.

IV. THE INFRASTRUCTURE TO SUPPORT DYNAMIC GRAPH PROCESSING

Most past work on dynamic graph colouring has focused on supporting a limited version of dynamic graphs where only a whole vertex can be added/deleted [33][34][35][36]. For colouring, this makes it trivial to employ the same greedy priority-based heuristic discussed in Section II: on a newly added vertex, simply gather the colours of its neighbours, then choose greedily the first colour available. On vertex deletion, the existing colouring remains always valid and thus this approach is still correct.

More importantly, a vertex-centric solution to support dynamic graphs does not realistically model graph evolution: in many cases, the evolution of the graph is edge-centric, that is, an edge may appear between two already established vertices, and a single edge may disappear (without removing any vertex). Our goal is thus to support an edge-centric view of graph evolution. (Note that building a graph vertex by vertex is actually a sub-case of our problem: we can simulate adding a full vertex by simply adding all edges associated with it all at once.)

Importantly, dynamically adding edges implies that we can no longer guarantee that a vertex can maintain its colour forever once it has been coloured. If a new edge is added between vertices that have the same colour, re-colouring must be triggered. The key implication is that in order to support online graph colouring for edge-centric graph evolution, the infrastructure must support cascading updates.

Our objective is to provide an event-based framework that ingests edge-centric graph structure changes and enables computation in a distributed manner by updating graph representation and generating the corresponding application-level events. The framework should handle application-generated events similar to push messages. This section discusses (i) how the dynamic graph is partitioned and stored in a distributed system to ensure load-balancing and performance while maintaining asynchronicity, (ii) how edge insertion and deletion are supported, and, finally, (iii) how the infrastructure supports online algorithm development.

A. Dynamic Partitioning and Storage

The first challenge we face is partitioning the dynamic graph over a distributed set of compute nodes. Since the graph is dynamic, a-priori there is no information that can inform partitioning, thus we choose a random partitioning technique. Our goal is thus to support an edge-centric view of graph evolution. (Note that building a graph vertex by vertex is actually a sub-case of our problem: we can simulate adding a full vertex by simply adding all edges associated with it all at once.)

Importantly, dynamically adding edges implies that we can no longer guarantee that a vertex can maintain its colour forever once it has been coloured. If a new edge is added between vertices that have the same colour, re-colouring must
Fig. 5: Webgraph: Run-time (on y axis, log scale) versus number of nodes (x axis), for multiple given an algorithm (Hash and LDF) and type of memory used (RAM and NVRAM). The right set of bars presents the total runtime using the dynamic algorithm (Section V). Note that runtime for collect&colour stages and the total time are presented separately on the left set of bars.

process can determine in constant time which process owns a vertex. Consistent hashing produces a balanced partitioning in terms of number of vertices, yet the resulting edge distribution may not be balanced: since we allocate vertices to processes, the power-law nature of many graphs may create an uneven balance with some processes responsible for vertices with many edges. For this prototype, however, we prefer simplicity; this naive strategy allows us to focus on the key issues related to supporting online algorithms and, importantly, provides a baseline lower bound for the performance that could be achieved using dynamic load balancing techniques in future work.

At the node level, we leverage DegAwareRHH, a high performance dynamic data-store designed for scaling out to store large, scale-free graphs by leveraging compact hash tables with high data locality [18]. DegAwareRHH follows an adjacency list representation and adopts an open address hashing to construct its vertex-table and edge-list.

B. Dynamic Graph Construction

The key advantage of the partitioning and storage model used is that any process is able to insert a new directed edge at any time. The directed edge will be co-located with the source vertex since the source vertex must know it has an edge to the destination. This makes it possible to support asynchronous directed edge creation and deletion.

The infrastructure supports splitting the stream of incoming graph update events among all the participating nodes. Thus in our experiments, each process independently ingests pairs of [source, destination] graph structure changes (edge events) from a text file, simulating a "stream".

Inserting or deleting undirected edges does require synchronization. Since an undirected edge creation/removal leads to state updates at two vertices, those updates must be coordinated to maintain a consistent view of graph state. To this end, we decided to "serialize" undirected edge creation: process P sends the creation of edge [a, b] to A, and then process A is responsible to send the create edge [b, a] to process B. Since the channel between processes A and B is FIFO, and only the two processes are able to use the edge, we thus ensure that the edge is created before it is used at both ends. Edge deletion is done in the same way.

C. Programming Abstraction

The infrastructure layer provides the programmer a set events that mirror graph evolution, for which the programmer can define callbacks to implement algorithm-specific functions. The programmer can also define new types of events and callbacks that use the same visitor model as discussed in section II-B. We focus here on the three events that need to be handled to support graph colouring for dynamic graphs:

Add/ReverseAdd Event: An edge addition triggers an Add event at the source vertex of a directed edge (or at the first vertex added for an undirected edge). An additional event, ReverseAdd, is used at the other vertex for an undirected edge.

Del/ReverseDel Event: Similar to the Add event, but when an edge deletion is encountered. Similarly, the subcase for ReverseDel exists.

Check Event: This event does not represent a structure change. It gives a callback processing a vertex the ability to generate events carrying vertex state changes to neighbouring vertices.

V. GRAPH COLOURING IN OUR INFRASTRUCTURE

We have developed a novel online graph colouring heuristic that supports edge-centric graph evolution. This heuristic defines the callbacks associated with the three abstract events previously defined, and is presented in detail below and in Algorithm 5.

Edge Add Event. For graph colouring, we add one additional optimization: since graphs are undirected, we can choose which process begins the two-step edge creation. We choose the vertex with the higher priority, since we know that if the two vertices have the same colour, the vertex with the higher priority will maintain its colour, and thus the second vertex can immediately recolour as seen starting on line 17 of
Algorithm 4: Dynamic Algorithm Infrastructure

```cpp
// Neighbour properties (e.g. 1 per edge).
map<int nbr_ID, int nbr_value> nbrs;
static Graph* graph;
static Queue* q;

void VISIT(int caller_ID, int caller_value, enum VISIT_TYPE) {
    switch (VISIT_TYPE) {
        case 'ADD':
            graph.insertEdge(caller_ID, this.ID, this.nbrs);
            add(caller_ID, caller_value);
            q.insert(caller_ID, this.ID, this.value, 'ADD);
            break;
        case 'REVERSE_ADD':
            graph.insertEdge(caller_ID, this.ID, this.nbrs);
            reverse_add(caller_ID, caller_value);
            q.insert(caller_ID, this.ID, this.value, 'REVERSE_ADD');
            break;
        case 'CHK':
            this.nbrs.set(caller_ID, caller_value);
            chk(caller_ID, caller_value);
            break;
        case 'DEL':
            int deleted_value = this.nbrs.get(caller_ID);
            del(deleted_value);
            graph.removeEdge(caller_ID, this.ID, this.nbrs);
            break;
        case 'REVERSE_DEL':
            int deleted_value = this.nbrs.get(caller_ID);
            del(deleted_value);
            graph.removeEdge(caller_ID, this.ID, this.nbrs);
            break;
    }
    q.insert(caller_ID, this.ID, this.value, 'CHK');
    q.insert(caller_ID, this.ID, this.value, 'REVERSE_ADD');
    q.insert(caller_ID, this.ID, this.value, 'REVERSE_DEL');
}
```

This highlights how the difference between Add and ReverseAdd events can be used advantageously.

**Edge Delete Event.** One could naively remove an edge and perform no update given that the colouring remains valid. However, this may result in a colouring that is further from ‘optimal’. In the original greedy approach on a static graph, where we know the degree of a vertex, we can assert that the given colour for a vertex is no larger than its degree plus one. This is because in the worst case all neighbours of the vertex are different colours, from the first colour to the degree colour; thus the vertex will choose the next available. However, in our dynamic case the degree is not known and, in fact, can drastically change during the lifetime of a vertex. As such, a vertex might end up with a large colour ID due to having a large number of higher priority neighbours, but then the vertex may lose these edges – thus being left in a state such that its colour is greater than its current degree, and potentially inflating the global colour count. To combat this, on edge deletion the two vertices both receive the notification, and each determine if they are able to recolour themselves with the colour the opposing vertex had. Thus, a smaller colour count is dynamically preserved. In this case, both callbacks corresponding to undirected edge delete events target the same function (line 47).

Check Event. When adding an edge, there are three possibilities: neither vertex previously exists, one vertex has already been created, or both vertices exist. The only non-trivial case is when both vertices exist and have the same colour, thus adding the edge generates a conflict. In this case one vertex will immediately recolour (as discussed above in the Add event).

Our choice of a parallel asynchronous model requires that we must impose a colour correctness check, as multiple connected vertices may need to recolour in parallel. An example is when we have two vertices \([A, X]\) already established with an edge, and in parallel \(A\) becomes connected to \(B\), and \(X\) to \(Y\); these can independently cause \(A\) and \(X\) to choose the same, new colour. The check event is used to deal with these cases: whenever a vertex chooses a colour it uses check to broadcast it to all its neighbours. If a conflict is detected, a vertex determines if it must recolour itself based on its Hash-priority. (Note that we do not use the degree-based priority, because the hash value remains constant but the vertex degrees may not). To summarize: the check event is used to announce new colour to neighbours; depending on priority, these may react and recolour themselves if a conflict is detected.

**Termination argument.** We briefly sketch a termination argument: In essence, add/delete events will generate Reverse-add/delete, and may generate check. Reverse-add/delete may generate check. Check may generate check, but only if the incoming check was from a higher priority vertex. As such, check at worst cascades down to the lowest priority reachable. In colouring, check generally stops at one hop (only local neighbourhood).

**Infrastructure support for optimized state storage.** In algorithm \([4]\) we use a map in line \([5]\) to store, for each vertex, a property for each of its neighbours. Conceptually a map is the right data-structure for this type of processing: although for each vertex, we know our (adjacency) list of neighbours, when we receive a message from a neighbour, we do not know the index of this neighbour. Although a map solves this issue by providing a way to later lookup the associated property (e.g., the key is the neighbour’s ID, the value is the property), we found this to become a performance overhead since nearly every message will access and update this data, from line \([21]\).

This “reverse lookup” from neighbour ID to associated value, becomes a bottleneck.

To alleviate this bottleneck we leverage infrastructure support. DegAwareRHH, the high-performance data-structure we use to store the graph locally, also provides the ability to store vertex properties and optimizes the “reverse lookup” mentioned. Our experiments in the next section highlight the gains enabled by using this infrastructure-level support.

**VI. ANALYZING DYNAMIC GRAPH COLOURING**

Using our previous static results as a baseline, this section shows the previously outlined dynamic colouring heuristic in action through our developed framework. We show scaling as before, and more importantly, results of our online algorithm on our curated dynamic real-world dataset.

**A. Experimental Environment**

The experimental platform is the same as in section \([III-A]\). We use the static graphs in Table \([II]\) with an arbitrarily
Algorithm 5: Dynamic Colouring

```cpp
class GraphColourVertex: public Vertex {
  // If we are a new vertex, colour us.
  if (this.value == 0) {
    this.value = 1;
  }

  void add(int caller_ID, int caller_value) {
    // If we can either have the first or second colour,
    // since we only have one edge to begin.
    if (this.value == (caller_value == 1) ? 2 : 1) {
      this.value = this.findFirstUnused(this.nbrs);
      check_nbrs(this.value);
      return; // A nbr has that colour; cannot take.
    }

    for (nbr : this.nbrs.iterator()) {
      if (deleted_value > this.value)
        return; // Only want to take a smaller colour.
    }

    check_nbrs(this.value);
    this.value = findFirstUnused(this.nbrs);
    // Conflict; we need to recolour.
    return; // We have priority, remain as-is.
  }

  void reverse_add(int caller_ID, int caller_value) {
    // Not new vertex, & we know we are lower priority.
    if (this.value != caller_value)
      return; // No conflict, remain as-is.
    if (priority(this.ID) > priority(caller_ID))
      return; // Have priority, remain as-is.
    // Conflict; we need to recolour.
    this.value = findFirstUnused(this.nbrs);
    check_nbrs(this.value);
  }

  void chk(int caller_ID, int caller_value) {
    if (this.value != caller_value)
      return; // No conflict, remain as-is.
    if (priority(this.ID) > priority(caller_ID))
      return; // Have priority, remain as-is.
    // Conflict; we need to recolour.
    this.value = findFirstUnused(this.nbrs);
    check_nbrs(this.value);
  }

  void del(int deleted_value) {
    if (deleted_value > this.value)
      return; // Only want to take a smaller colour.
    for (nbr : this.nbrs.iterator()) {
      if (nbr.value == deleted_value)
        return; // A nbr has that colour; cannot take.
    }

    this.value = deleted_colour;
    check_nbrs(this.value);
  }

  void reverse_del(int deleted_value) {
    del(deleted_value);
  }
}
```

generated edge creation ordering, as well as the dynamic Wikipedia corpus detailed in Table III with true dynamic ordering.

When analyzing performance for colouring dynamic graphs and comparing it with the performance of the equivalent static graph, there are important differences to keep in mind. First, in the static case we start the timer after the graph has loaded into memory. This implies some pre-processing effort is not included in reported timing. For example, after loading, we know the properties of the graph such as the degree of each vertex, etc. In the dynamic case, we no longer have this information: the performance is tied to NVRAM performance, and the algorithm has no understanding of the future. We build the graphs from nothing, by design: each process reads edges from NVRAM one at a time, executing as a stream of add edge events. (Delete events are used in the dynamic Wikipedia dataset only).

In addition, note that for dynamic graphs the colour count is no longer deterministic as in the static case, and depends on the order of communication among processes. Due to this, dynamic colour count results are also averaged over the 10 executions; however, the result was surprisingly stable.

B. Comparison to Static Solution Quality: Colour Count

To determine the quality of the solution offered by the dynamic algorithm, we ran our dynamic algorithm on the same static graphs as before. In terms of colouring, two metrics are important: Foremost, colour count – a solution using more colours is worse. Secondly, the distribution of chosen colours will show how similar solutions are. In Figure 6 we show an overlap of the colour distributions offered by the static Hash method, and the dynamic one. Notably, for SK2005 the two solutions use the same number of colours. For Friendster the dynamic solution needs ~15% more colours, for Twitter ~16%, and for the Webgraph less than 1% more. However, despite occasionally using slightly more colours, the distributions of the colours are remarkably similar (represented by the green overlap in the figures). Interestingly, the most noticeable difference is the set of vertices that chose the first colour: for dynamic graphs, it was around half or fewer vertices as compared to the static case, which likely caused the increase in colour count. In general, the colour count and distributions are remarkably similar, showing that the solution space itself is similar.

C. Reasoning about Performance

In terms of performance, as mentioned before there are a multitude of reasons why the dynamic algorithm has largely worse performance than the static one (as is with most online algorithms). Of course, the algorithm itself is different, leading to multiple messages across edges as colours evolve and change. However, there are two additional important factors: inherent graph storage inefficiencies, and performing graph construction online while colouring.

Graph Storage. The static analysis can use efficient, compressed storage formats (e.g., Compressed Sparse Row - highly memory and access efficient) as the final graph structure is known at load-time. This is impossible for the dynamic case. Data presented in Figure 4 provides a first-order estimate of this impact: we show the difference between using a map as a vertices’ edge list (i.e. adjacency list), versus DegAwareRHH (see section IV-A). Notably, using a map (we used a std::unordered_multimap) is not memory efficient – ending up requiring over 200GB of memory for any of the three graphs. As such, it would not fit into a single node, but does provide a baseline for DegAwareRHH.

Graph Construction. Secondly, the dynamic algorithm and storage infrastructure do not know how large the graph will grow to be, nor any properties of the graph. As the edge events are streamed in to build the graph in-memory, read-speed becomes intermixed into the processing time as well. In Figure 5 the Webgraph shows results for statically computing from NVRAM, showing similar performance to the dynamic algorithm, due to the high read I/O load. Finally, we note an extra overhead present in the dynamic case versus the static one: edges, when read, are redistributed with probability \((N - 1)/N\) to another one of the \(N\) nodes. In the static case, the partitioning is known before algorithm execution, and the graph is not written, only read.
Fig. 6: Colour Distribution Analysis: number of vertices [y-axis, log scale] choosing each colour [x-axis]. The bars of dynamic [yellow] and static (Hash method) [cyan] are overlapped of each other, forming green for overlap. For all graphs, on colour 1, there is a large amount more static [cyan], although this is difficult to see. More importantly, the visible yellow part implies a poorer solution, i.e., more colours used, for dynamic colouring.

D. Performance and Scaling

Figure 7 shows the result of running the dynamic algorithm to completion on the same real-world static graphs as in Figure 4 ingesting one parallel stream per process (24 processes per node). The Hash method and GraphLab results are also shown. Surprisingly, the scaling for dynamic executions is good, and actually performs faster than the static GraphLab at higher node counts. The key reason for good scaling is that, as discussed previously, in our environment the performance of the dynamic execution is limited by the read rate of the input stream, i.e., the NVRAM read IO. By doubling node count, we not only double processor power, but we also double read throughput.

The key takeaway. Figure 7 highlights that, not only the cost of an online solution is not prohibitive, but it is likely more efficient to use an online solution than to snapshot the graph and process snapshots with a static solution, despite previous work targeting this direction [38]. For high node counts, the online solution is only 2-4x slower than our best static solution (and 3-10x faster than the static GraphLab solution), despite discounting static pre-processing. Even at the much larger scale of the Webgraph (Figure 5, right bars) the same relationship holds: the data suggests that an online algorithm becomes the optimal solution even after only a dozen snapshots, and offers much finer granularity than snapshots would!

E. Real Dynamic Graph: Wikipedia Dataset

To test our algorithm at scale with real-world data, we have curated the largest dynamic graph data set, to our knowledge, from over a decade of the English Wikipedia corpus. Vertices are pages, and edges are hyperlinks with start and end time stamps corresponding to the edge creation and possible deletion events. General statistics about the Wikipedia dataset are outlined in Table III. To curate this dataset, we started with the full historical XML dumps provided by Wikipedia, and parsed the wiki markup to extract internal (intra-wiki) and external hyperlinks. This process extracted of the hyperlinks from all historical revisions from all English pages. Consecutive revisions of a page are compared to check if links were created or deleted. All hyperlink creation and deletion events from all


<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time Range</td>
<td>Jan 2001 - Dec 2015</td>
</tr>
<tr>
<td># Edge Adds</td>
<td>2,713,888,893</td>
</tr>
<tr>
<td># Edge Deletes</td>
<td>1,806,190,225</td>
</tr>
<tr>
<td># Unique Vertices</td>
<td>205,774,846</td>
</tr>
<tr>
<td># Unique Edges</td>
<td>1,303,659,380</td>
</tr>
</tbody>
</table>

Some applications of dynamic graphs may involve multiple parallel streams of edge operations. To evaluate the performance of our algorithms when processing parallel streams, we split the single Wikipedia edge stream into multiple parallel streams. The split occurs by hashing each edge’s source vertex to select a stream. It is important to note that our parallel streams will not preserve the exact ordering of the single stream, but serve as a realistic test for performance evaluation.

Figure 8 shows the scaling performance of dynamic colouring on the Wikipedia dataset with varying number of edge streams (1-64). Scaling a single sequential edge stream is limited to a few compute nodes; however, the algorithm scales well when parallel streams are used, which matches real world concurrent updates from multiple sources. Figure 9 shows the actual results of the dynamic colouring algorithm, with a granularity of years selected to be shown – displaying the evolution over time.

VII. RELATED WORK

Dynamic Graphs: Online Algorithms and Infrastructure Support. Graphs are widespread data structures used to model a wide variety of real-world systems. The sheer amount of data to be processed has prompted the creation of a myriad of systems to cope with massive scale graphs. Two additional factors drive our work: firstly, the increasing pressure to deliver fast responses demanded by many applications (e.g., online recommendations, auctions, terrorism protection), and, secondly, the need to model continuously evolving real world systems represented as graphs (e.g., social networks). Thus graph processing system must support both requirements: near real-time analysis, and dynamic massive graphs.

To date, the main approach towards supporting these requirements has been snapshotting the graph and using existing
Fig. 7: Run-time for colouring real-world (SK2005, Friendster, and Twitter, from left to right) for static (Hash and GraphLab) and dynamic methods. The dynamic colouring methods include a baseline map graph storage (MAP), and DegAwareRHH (DARHH). Run-time on [y axis, log scale], versus number of nodes [x axis]. Note that y-axes start at different points. (To guide interpretation, the diagonal lines in grey indicate perfect strong scaling). We do not show speedup as: (i) a one-node baseline does not exist in many cases, and (ii) we believe absolute runtime, and not speedup, are more informative.

Fig. 8: Wikipedia Dataset: Number of nodes [x axis] compared to Edge Operations (e.g. add/delete) per second [y axis, millions, log scale]. The plot presents number of parallel streams (how many of the events are processed in parallel).

Fig. 9: Wikipedia properties over time (obtained with the online algorithm): Shows the evolution of the Wikipedia graph as edges are created and deleted. Vertex/Edge counts [y axis] and Colour counts [shown in boxes] for each year.

Graph analytics tools designed for static graphs to cope with massive scale. The existence of a potentially more efficient avenue that avoids snapshotting, however, is suggested by past work on online algorithms for dynamic graphs [39][40]. These algorithms dynamically maintain a (full or partial) solution to the user query and update it as the graph evolves. The infrastructure support for such online solutions, however, has seen little past work [9], and none has been carried out to implemented systems that can scale to analyze massive graphs.

Colouring Dynamic Graphs. We are the first to offer a generic, distributed, online graph colouring algorithm. Our solution is generic in the sense that it is constraint free, and supports adding or deleting any edge at any time. Most past solutions to colour dynamic graphs [33] [34] [35] and [36] are vertex-centric and assume that each new vertex is given together with all [its] edges joining it to previous vertices. Other solutions [41] assume that the set of vertices is static (no new vertices are inserted or old ones removed.) and only specific edges can be modified. Furthermore, none of these solutions are presented as parallelizable, let alone distributed.

Colouring Static Graphs on Distributed Platforms. Bozdag et al. [27] present a colouring solution for distributed memory platforms. This work, however, uses unrealistic workloads of tiny graphs - their largest graph is 10,000 times smaller than the graphs used here, with a node degree distribution that is uniform. Gandhi et al. [28] use Hadoop to colour graphs; they experiment on even tinier workloads. Hansen et al. [29] present a distributed largest-first algorithm, but do not offer performance results. Salihoglu et al. [30] present a colouring algorithm for Pregel-like systems and evaluate it on Friendster, Twitter, and SK2005 real-world graphs as we do (yet they do not demonstrate scalability as their largest workload is 60x smaller than ours). The reported processing times on 90 nodes are: 26.4 minutes for SK2005, 5.9 minutes for Twitter, and 8.5 for Friendster. Lu et al. [31] also colour the Friendster graph in around 10 minutes, yet on 15 nodes. Granted our nodes are more powerful, but the system we developed is much more efficient as well: we can colour all these graphs in under a minute on a single node; achieve excellent scaling (Figure 4), and colour a graph 60x larger...
in under two minutes (Figure 5).

**Colouring Static Graphs on Shared Memory Platforms.**
Most of the past experience with graph colouring is on shared memory platforms, yet uses workloads of limited scale [42-43]. More recently, Catalyurek et al. [44] present algorithms designed for many-core platforms and explore the algorithm-architecture interplay. They report results for small scale workloads on three platforms: Cray XMT, Sun Niagara and Intel Nehalem. Rokos et al. [32] focus on massively parallel accelerators, yet this limits the scale of the graphs they can process: they only use graphs with up to 128M edges (2,000x smaller than our largest workload). Most related to our work, FlashGraph [13], demonstrates the ability to process massive graphs on powerful single nodes with ample NVRAM space: they use the same Webgraph in experiments, with a much larger single node (2x more CPUs, 4x larger memory space, 15x higher SSD count). Their architecture is comparable to HavoqGT if only considering the single-node level.

**VIII. CONCLUSION**
This paper uses graph colouring as the challenge problem to explore the design space of infrastructure-level support for online analytics for dynamic graphs. Our framework efficiently handles graph evolution at the edge-level, is asynchronous and distributed, and leads to impressive performance and scalability.

The properties of the novel online colouring algorithm we design and evaluate suggests that this exploration path can be fruitful and deserves further attention: firstly, our experiments show that, although inherently leading to higher overheads, an online algorithm quickly becomes the most attractive solution compared to a static algorithm that takes graph snapshots from a dynamic trace. Secondly, our experience shows that it is possible to hide most of the development complexity in the infrastructure, and provide the algorithm developers with a simple, event-based interface.

Finally, we curate a new dataset – a decade of the English Wikipedia corpus – with true dynamic properties of time-dependent additions and deletions of hyperlinks. We show that (i) our infrastructure supports true dynamism in graphs, and (ii) we can achieve tangible, and live (with no knowledge of future events) algorithmic results, which we show in our challenge problem of dynamic graph colouring.

**ACKNOWLEDGEMENT**
This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under contract DEAC52-07NA27344 (LLNL-CONF-688697). Funding was partially provided by LDRD 16-PS-024 and by the support of the University of California through the UC Laboratory Fees Research Program (12-LR-236705). Experiments were performed at the Livermore Computing facility.

**REFERENCES**


