Exploring Accelerator and Parallel Graph Algorithmic Choices for Temporal Graphs

Akif Rehman
akif.rehman@uconn.edu
University of Connecticut
Storrs, CT USA

Masab Ahmad
masab.ahmad@uconn.edu
University of Connecticut
Storrs, CT USA

Omer Khan
khan@uconn.edu
University of Connecticut
Storrs, CT USA

ABSTRACT

Many real-world systems utilize graphs that are time-varying in nature, where edges appear and disappear with respect to time. Moreover, the weights of different edges are also a function of time. Various conventional graph algorithms, such as single source shortest path (SSSP) have been developed for time-varying graphs. However, these algorithms are sequential in nature and their parallel counterparts are largely overlooked. On the other hand, parallel algorithms for static graphs are implemented as ordered and unordered variants. Unordered implementations do not enforce local or global order for processing tasks in parallel, but incur redundant task processing to converge their solutions. These implementations expose parallelism at the cost of high redundant work. Relax-ordered implementations maintain local order through per-core priority queues to reduce the amount of redundant work, while exposing parallelism. Finally, strict-ordered implementations achieve the work efficiency of sequential version by enforcing a global order at the expense of high thread synchronizations. These parallel implementations are adopted for temporal graphs to explore the choices that provide optimal performance on different parallel accelerators. This work shows that selecting the optimal parallel implementation extracts geometric performance gain of 46.38% on Intel Xeon-40 core and 20.30% on NVidia GTX-1080 GPU. It is also shown that optimal implementation choices for temporal graphs are not always the same as their respective static graphs.

CCS CONCEPTS

• **Computer systems organization → Multicore architectures**; • **Computing methodologies → Shared memory algorithms**.

KEYWORDS

multicores, graph algorithms, static graphs, temporal graphs, performance scaling

ACM Reference Format:

Akif Rehman, Masab Ahmad, and Omer Khan. 2020. Exploring Accelerator and Parallel Graph Algorithmic Choices for Temporal Graphs. In *The 11th International Workshop on Programming Models and Applications for Multicores and Manycores (PMAM'20), February 22, 2020, San Diego, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3380536.3380540

PMAM'20 , February 22, 2020, San Diego, CA, USA

ACM ISBN 978-1-4503-7522-1/20/02…$15.00
https://doi.org/10.1145/3380536.3380540

1 INTRODUCTION

Graph algorithms are widely utilized in real world due to their enormous number of applications in different domains, such as traffic map applications [4], self-driving cars [33], social network analytics, and routing algorithms [24]. From the perspective of graph theory, given a graph G (V, E) with $|V|$ vertices and $|E|$ edges, a graph algorithm performs computations on the vertices and its corresponding edges, and returns some specific output.

Conventionally, graph algorithms are designed for input graphs that are static in nature. The properties of static graphs don't evolve with time, i.e., the number of vertices, edges, and weights of the edges are not a function of time. However, many real-world applications encounter graphs that are time-varying, i.e., their characteristics change over time. Graphs emerging in different domains, such as traffic predictions [12], social networks [30], biological sciences [6], and wireless sensor networks [25] are all examples of such time-varying graphs. In these graphs, edges appear and disappear with time and the weights of the edges also fluctuate with respect to the time [16]. Due to the omnipresent applications of time-varying graphs in various domains, it is important to study parallel implementations, and performance scaling of different graph algorithms using time-varying graphs.

Prior research on time-varying graphs focused mostly on representation and modeling [16, 27, 36], and the analysis of temporal graphs [5, 7, 32, 35]. On the algorithmic front, traditional graph algorithms, such as the single shortest path problem [11, 13, 16, 37, 38], breadth and depth first search [21], minimum spanning tree [22], page-rank [20, 29], and community detection [19] are proposed for temporal graphs. However, performance analysis of parallel algorithms in terms of optimal implementation (parallel implementation that gives best performance) on different accelerators (such as a multicore or a GPU) tends to be overlooked in prior literature. This is because the purpose of the aforementioned works is not to parallelize the already available graph algorithms, but to create sequential algorithms for temporal graphs that are analogous to the existing algorithms for their static counterparts. The absence of performance implication studies from prior literature is also due to the limited availability of temporal graph datasets. Additionally, the available temporal graphs are smaller in size [28] as compared to their static counterparts, hence limiting the opportunities for exploiting parallelism.

In order to study performance of temporal graph algorithms on different accelerators, there is a dire need for temporal graphs with reasonable number of vertices and edges. Instead of generating such temporal graphs from scratch, static graphs (such as California road network (USA-CAL) [10] or biological graph (Cage14) [28]) can be converted into temporal graphs. This motivates the need

for a temporal graph generator that takes a static graph as input, and outputs its temporal version by systematically adding edges at specific time instants to represent a temporal graph. The graph generator varies the weights of edges at different time instants as the output of some graph algorithms may also be sensitive to the value of weights.

The conversion of real static graphs into temporal graphs provides an opportunity to explore the performance scaling of various temporal graph algorithms on different parallel accelerators. The parallel implementations of these graph algorithms can be categorized based on the task ordering constraints under the task-parallel execution model. The unordered parallel implementations of these algorithms remove the execution order altogether, and expose parallelism at the expense of high redundant work. Alternatively, the relax-ordered implementations only enforce local order through per core priority queues to reduce the amount of redundant work. In order to achieve the work efficiency of their sequential counterpart, strict-ordered implementations maintain global order using queueing primitives that impose high thread synchronizations. The benchmark algorithms executing temporal graphs are evaluated in the context of their performance scaling on a given parallel machine.

The optimal implementation for a temporal graph algorithm and input is the one that gives the best performance on a parallel accelerator. A certain temporal benchmark-input combination may give superior performance on a multicore and another benchmark-input combination may prefer a GPU [1]. This motivates the need for exploring the optimal implementation and accelerator choice space for a given temporal benchmark-input combination. The contributions of this work are outlined below:

- A temporal graph generator is proposed and implemented that transforms static graphs into temporal graphs. This allows to explore the performance aspects of graph algorithms executing temporal graphs.
- Various unordered and ordered parallel temporal graph algorithms are developed and benchmarked.
- The optimal performance scaling choices for temporal graph benchmarks and inputs on different parallel multicore and GPU accelerators are explored. It is shown that selecting the optimal implementation rather than always selecting strict-ordered implementation for the Intel Xeon 40-core machine gives a geometric performance gain of 46.38%. On an NVidia GTX-1080 GPU, selecting the optimal implementation gives 20.3% performance advantage as compared to always selecting the relax-ordered implementations.
- The optimal choice of an accelerator for temporal graph benchmarks and inputs is also studied. It is shown that selecting an optimal accelerator for a given benchmark-input combination leads to a geometric performance gain of 30% as compared to always selecting the GPU machine.

2 RELATED WORK

Many real-world applications utilize graphs that are time varying in nature. Traffic predictions [12], social networks [30], biological sciences [6], and wireless sensor networks [25] are few examples where temporal graphs are used. Representation and modeling of

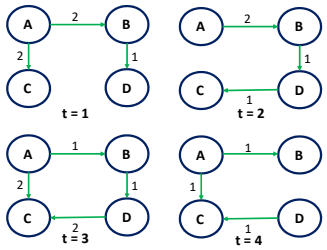

Figure 1: Snapshots of a directed and weighted temporal graph at four time instants

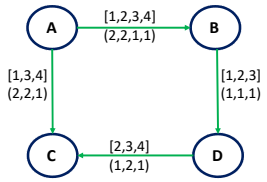

Figure 2: Time aggregated graph representation of a temporal graph

time varying graphs has been given a lot of attention in prior literature [16, 27, 35, 36]. The Chronos [17] graph engine proposes a layout scheme for temporal graphs that utilizes the structural and temporal locality to efficiently process these graphs. Several conventional graph processing algorithms, such as shortest path, traversal, and spanning tree algorithms are also developed for temporal graphs [20, 21, 37, 38]. Even though there has been research on the representation and algorithmic front, the performance analysis of different parallel implementations for temporal graph algorithms is lacking in literature. There are three distinct categories of parallel implementations for static graph algorithms, namely strict-ordered [18], relax-ordered [26] and unordered implementations [2]. For static graphs [14, 23], the performance of an implementation is dependent on the variations in graph algorithms and inputs [1]. Consequently, the selection of an optimal implementation on a given parallel machine is also important for temporal graphs and algorithms.

3 TEMPORAL GRAPH REPRESENTATION

In a time-varying or temporal graph, the existence of an edge is dependent on time, i.e., edges appear and disappear at different time instants. The weights of edges are also a function of time and have different values at different time instants. Figure 1 shows an example of a directed and weighted time-varying graph at four different time instants. This graph contains four vertices connected through time-dependent edges. At different time instants, the total number of edges in this temporal graph vary as the edges appear and disappear in time. For example, the edge going from node A to node C is present at time instants 1,3 and 4, and is missing at time instant 2. Additionally, the weights are also changing in this temporal graph. For example, the edge going from node A to node B has a weight of 2 at time instants 1 and 2, and a weight equal to 1 at time instants 3 and 4.

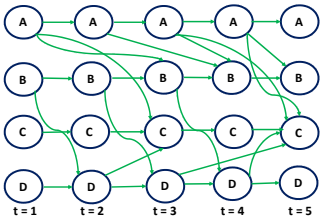

Figure 3: Time expanded graph representation of a temporal graph

In order to represent different snapshots of a temporal graph in one concise representation, George and Shekhar proposed time-aggregated graphs [16]. Time aggregated graphs represent temporal graphs by capturing their characteristics through time series. There are two series associated with each edge of a temporal graph. The first series contains the values of time instants for which the edge is present, and the second series contains the weights of the edges at those time instants. Figure 2 displays the time-aggregated graph representation of the temporal graph shown in Figure 1. Each edge in the graph is represented with two series. The series in [ ] contains the value for time instants, while the series in ( ) contains the weights at those time instants.

An alternative representation of temporal graphs is to expand the time dimension. This can be done by replicating every node at each time instant. This representation of the temporal graphs is called time-expanded graph. The weights of all the edges in time expanded graphs are equal to 1. Figure 3 displays the time-expanded graph representation of the temporal graph shown in Figure 1. Every node in this graph is repeated five times in the figure. This is because of the edge going from node A to node B at time instant 4 having a weight equal to 1. The maximum value of the sum of the time instant at which the edge is present and weight of the edge across all edges defines how many times every node is replicated. For the time-expanded graph shown in the figure, the maximum sum is 5.

Due to the fact that nodes are replicated in time expanded graphs, they take large amount of memory as compared to the time-aggregated graphs. The total number of nodes and edges in the time-expanded graphs are also greater than the number of nodes and edges in time-aggregated graphs. This means that a graph algorithm performs more work for the time-expanded graph as compared to its time-aggregated counterpart. As the redundant work can immensely affect performance of a parallel implementation, time-aggregated graphs tend to be a better and compact way of representing the temporal graphs. Therefore, time-aggregated graphs are utilized in this paper.

4 TEMPORAL GRAPH GENERATION

This section presents a method to convert a static graph into a temporal graph. Any static graph can be viewed as a temporal graph having all edges defined for only one time instant. This means that a static graph can be converted to a temporal graph by adding edges at different time instants, and changing the weights of edges for various time instants.

One simple and naive method to add temporal edges to a static graph is to randomly decide whether an edge should be added for a specific time instant or not. Algorithm 1 shows the pseudocode for temporal graph generation utilizing this simple strategy. The algorithm takes as input the static graph and the maximum number of time instants (T) for an edge, and outputs a temporal graph. It has an outer loop that goes over all the vertices of the static graph, and an inner loop that goes over the edges belonging to the vertex from the outer loop. There is another inner loop that goes over all the time instants and for each iteration of this inner loop, a number is randomly picked between 0 and 1. If this randomly picked number is greater than a certain threshold, then the edge is added for that time instant, otherwise not. The weight of all temporal edges are same as the weight of the static edge. The $threshold$ variable controls the number of temporal edges. For example, setting it to 0.5 adds approximately $T/2$ temporal edges for each static edge.

Algorithm 1 Naive algorithm for temporal graph generation

Inputs: $G(V, E)$ ← Static Graph, T ← Maximum time instants for an edge
Outputs: $G^t(V, E^t, t)$ ← Temporal Graph

```
1: for (each vertex v ∈ V) do
2:     for (each edge (v, u) of v with weight w) do
3:         for (each t in range (1,T) ) do
4:             num = sample_from_uniform_distribution (0,1)
5:             if (num > threshold) then
6:                 G^t.add_edge(v, u, w, t)
```

Algorithm 1 generates a temporal graph using the simple method. However, the generated temporal graph does not imitate a real temporal graph. The key shortcoming is that the algorithm treats edges belonging to different vertices equal since it always samples from the uniform distribution between 0 and 1. Additionally, all time instants are treated equally, and the weight of a temporal edge is the same as that of a static edge. However, in a real temporal graph this is not the case as there are varying patterns for different parts of the graph and different time instants. In order to put this into perspective, let's take example of the California road network graph. The traffic patterns for the big cities, such as San Francisco are different from the traffic patterns of a suburb (e.g. Pleasanton). Similarly, the traffic patterns during the rush hours are different than the patterns during other hours of the day. Due to this, the weights of edges in a temporal graph should increase during rush hours, and decrease during normal hours. There is a need to change Algorithm 1 in a way that for different parts of a graph, and for different time instants, sampling is done from a different probability distribution.

Algorithm 2 shows the pseudocode for an improved version of algorithm 1 for temporal graph generation. The graph is divided into n sets containing equal number of vertices. This is done to spatially distribute the graph into different parts. The partition is done based on the vertex ID, but if any additional information is available with the graph that can be used to achieve this partition. For example, the California road network graph can be divided into different parts based on the available longitude and latitude values. Time instants are also divided into m equal parts, where $m < T$. The loops for vertices, edges, and time instants remain the same as in Algorithm 1. However, for every vertex, the algorithm determines

Algorithm 2 Improved algorithm for temporal graph generation

Inputs: $G(V, E)$ ← Static Graph, T ← Maximum time instants for an edge
n ← number of vertex's sets, m ← number of time intervals
Outputs: $G^t(V, E^t, t)$ ← Temporal Graph

```
1: for (each vertex v ∈ V) do
2:     for (each edge (v, u) of v with weight w) do
3:         s = v % n
4:         for (each t in range (1,T) ) do
5:             ti = t % m
6:             w_scale = sample_from_probability_distribution(s, ti)
7:             w_t = w * w_scale
8:             G^t.add_edge(v, u, w_t, t)
```

its corresponding set, and for every time instant the algorithm determines the time interval it belongs to. The values of set and time interval are passed to the *sample_from_probability_distribution* method so that it can sample from the associated probability distribution. As there are n set of vertices and m set of time instants, there are a total of $m * n$ probability distributions. The probability distributions can be specified by the user based on their data. For the graph generated in this work, unit Gaussian distributions are used with different scaling factors. After sampling from the distribution, a scaling factor is obtained that is multiplied with the weight of the static edge to obtain the weight for the temporal edge. An edge is added to the temporal graph going from vertex v to u at time instant t with weight w_t.

The proposed temporal graph generation method is used to convert several static graphs from open source datasets into their temporal graph variants. The static graphs range from different sizes, diameter, and density. The generated temporal graphs and their different characteristics are outlined in the methodology section. The source code for the graph generator is released to public using a GitHub[1] repository.

5 PARALLEL TEMPORAL GRAPH ALGORITHMS

Exploiting task-level parallelism has gained popularity due to its integration in modern parallel programming frameworks [18, 26]. In order to expose parallelism, the parallel implementation of an algorithm specifies a unit of execution, called task that performs some fixed operations and executes in parallel with other tasks. The framework or the library provides various primitives to handle the low-level system operations, such as load balancing and synchronization. Task parallel workloads show superior scalability at higher core-counts as their execution model is independent of the underlying machine. A task execution order is present in all task parallel algorithms. The order is required as the execution of a particular task depends on the execution of other tasks, and synchronization is required to properly forward the inter-task dependencies. Different tasks can also operate on the data shared among tasks requiring locking mechanisms so that read-write dependencies are met. Due to this, thread synchronization becomes an important component of the execution model. There are no dependencies (inter or intra) in an ideal task parallel workload, and every task is free to execute in parallel with other tasks.

[1]https://github.uconn.edu/omk12001/TemporalGraphGenerator

Algorithm 3 Generic pseudocode of the strict-ordered implementation

tid ← Core ID
$PQ[tid]$ ← Priority Queue for each core
$TaskList$ ← Global Ordered List

```
1: for (each task in PQ[tid]) do
2:     task = PQ[tid].peek()
3:     test = safe_source_test()
4:     if (test = pass) then
5:         task = PQ[tid].pop()
6:         remove_entry_atomic(TaskList(task))
7:         for (each child of task do
8:             for t in child.T do
9:                 task = PQ[tid].push() or send_to_remote_core()
10:                critical_section(shared_data)
11:                add_entry_atomic(TaskList(child))
```

Strict-ordered algorithms maintain stringent ordering constraints on task execution. Due to this they have the work efficiency of their sequential counterparts. Extracting parallelism while following a strict global order is an extremely difficult problem. The Kinetic Dependence Graph (KDG) [18] framework is one such example that exploits task-level parallelism while enforcing strict task execution order. The local order in KDG is maintained through per-core priority queues, and the global order is enforced through a shared data structure (an ordered list) that orders task insertion and deletion. When a task is dequeued from the priority queue of a core, each core runs the safe-source-test that finds the dependency of the dequeued task on the tasks in other cores by looking up the ordered list. The dequeue operation stalls if a dependency is detected. However, tasks execute in parallel if no dependencies are found. The goal is to find tasks that can be executed in parallel on different cores to exploit parallelism. Algorithm 3 shows the pseudocode for a generic strict-ordered implementation of a representative graph algorithm. A local queue is implemented in each core, and there is a global ordered list shared among all cores. Each core looks for the highest priority task from its queue, execute the safe source test to make sure that there are no task dependencies. If the safe source test indicates that the task is independent, then the task is removed from the local priority queue and atomically deleted from the ordered list. Once the dequeue operation is complete, the implementation goes over all the children of the task, and then for all the time instants of each child to create new tasks. These tasks are either inserted into the local priority queue or communicated to a remote core for load balancing purposes. After the execution of critical section, each task is atomically added to the shared ordered list.

Parallelism is limited in strict-ordered algorithms due to the presence of global order. and thus high synchronization cost. Alternatively, large amount of parallelism can be exposed by performing redundant work. This approach is adopted by the Galois [26] framework by removing the global ordering constraints enforced by strict-ordered implementations of KDG, and having local priority queues per core for local order of task processing. As there is no order list shared among the cores, the synchronization cost is reduced in comparison to strict-ordered implementations. However,

Algorithm 4 Generic pseudocode of the relax-ordered implementation

```
tid ← Core ID
PQ[tid] ← Priority Queue for each core
1: for each task in PQ[tid]) do
2:     task = PQ[tid].peek()
3:     test = local_test()
4:     if (test = pass) then
5:         task = PQ[tid].pop()
6:         for each child of task do
7:             for t in child.T do
8:                 task = PQ[tid].push() or send_to_remote_core()
9:                 critical_section(shared_data)
```

the amount of redundant work is high in relax-ordered implementations since they may need to pass through multiple iterations of processing a task to converge on a solution. Algorithm 4 shows the pseudocode for a generic relaxed-ordered implementation of a representative graph algorithm. It has a local priority for each core but no ordered list as there is no global order. Additionally, safe-source test is replaced by a local test. Each core looks for the highest priority task in its queue and runs a local test on that task. If the test passes, the task is dequeued from the local queue. After the task is removed from the local queue, outer loop goes over the children of the task, and inner loop goes through time instants of each child. The created tasks are either inserted into the local priority queue or communicated to a remote core.

Algorithm 5 Generic pseudocode of the unordered implementation

```
tid ← Core ID
TaskList[tid] ← Local Work List for each core
1: for each task in TaskList[tid]) do
2:     task = TaskList.peek()
3:     task = TaskList.pop()
4:     for each child of task do
5:         for t in child.T do
6:             task = TaskList.push() or
7:             critical_section(shared_data)
```

Unordered task parallel algorithms remove both the local and global task execution order. They allow different tasks to execute in parallel with other tasks, and implemented in such a way that inter-task dependencies are minimized. Due to the absence of local and global order, unordered implementations pass through large number of iterations over the input graph before they converge. The number of iterations in turn increase the amount of redundant work in unordered implementations. The unordered implementations with reasonable work efficiency show significant scalability at higher core counts, and tend to provide superior performance as compared to their relax-ordered and ordered versions. Due to the read-write data dependencies in an unordered algorithm, it also requires some kind of synchronization. Even when the unordered implementation doesn't have read-write data dependencies, its different phases are separated by barriers. This is needed to ensure that all data dependencies are forwarded properly, and the next phase of the implementation can be safely initiated by each thread.

Algorithm 5 displays the pseudocode for a generic unordered implementation of a task parallel workload. Each core has an unordered list (instead of priority a queue) that contains the tasks to be executed. Tasks present in the task list of different cores are be executed in parallel. Each core gets a task from its task list, then runs a loop over all the children of the task, and the time instants of the children tasks. For each iteration of the inner loop, it atomically performs some operations on the shared data and pushes the children to its local task list.

There are some tradeoffs between the parallelization strategy, synchronization, and work efficiency in the three implementations of a graph algorithm:

- Vertices or edges are statically distributed among the threads when the local and global order is not present. Large amount of parallelism is exposed in this parallelization strategy but the redundant work is also high.
- In the presence of a local order enforced through per core queues, vertices/edges are dynamically distributed among cores. This reduces the amount of redundant work while exposing significant parallelism.
- Optimal work efficiency can be achieved by maintaining a global order at the expense of high synchronization cost. However, this reduces the amount of exploitable parallelism.

The tradeoffs depict that the three implementations are different from each other in terms of parallelization strategy, synchronization, and work efficiency. These high-level characteristics of the implementations and temporal graph inputs are the primary source behind the optimal implementation and optimal accelerator selection. In the evaluation section, optimal implementation and accelerator choices for various temporal benchmark-input combinations are shown, and tied back to these high level-characteristics of the temporal graph algorithms.

The inner loop that goes over the time instants of a child is added to the different implementations of static graph benchmarks to convert them to their temporal counterparts. Due to the presence of these time instants in the inner loop, the amount of exposed parallelism increases. As there is higher parallelism in temporal implementations, the choice of optimal implementation for temporal algorithms and graphs may not be same as their static counterparts.

6 METHODOLOGY

6.1 Machine Configurations

Inter-implementation choices for different static and temporal benchmark input combinations are evaluated on a multicore and a GPU machine. Intel Xeon E5-2650 v3 is used as a multicore machine. It has 10 hyper-threaded cores clocking at 2.30 GHz and a 1 TB DDR4 RAM. The NVidia GTX-1080 GPU is used for evaluation as well. The GPU has 2560 cores with 8.8 $TFLOPs$ single-precision, and 0.27 $TFLOPs$ double-precision compute capability, and has 8 GB memory size. Table 1 shows different parameters of these parallel machines.

Table 1: Accelerator Configuration.

	GTX-1080	Xeon E5-2650 v3
Cores, Threads	2560, Many	10, 40
Cache Size, Coherence	2MB, No	25MB, Yes
Mem. (GB), BW. (GB/s)	8, 320.3	1000, 68
Single-Precision (TFlops)	8.8	2.8
Double-Precision (TFlops)	0.27	1.4

Table 2: Benchmark categorization based on available implementations

Strict, Relax and Unordered	Unordered only
Single Source Shortest Path (SSSP)	Depth First Search (DFS)
Breath First Search (BFS)	PageRank (PR)
Connected Component (CC)	PageRank-DP (PR-DP)
Minimum Spanning Tree (MST)	Triangle Counting (TC)
Graph Coloring (Color)	Community (Comm)
Astar search (A*)	

6.2 Benchmarks and Inputs

Graph algorithms utilized in this paper fall into two distinct categories. There are three different implementations namely strict-ordered, relax-ordered, and unordered for the benchmarks that belong to the first category. In the second category, the benchmarks only have unordered implementations. Table 2 displays the graph benchmarks that belong to these two categories. The inter-implementation choices are only valid for the benchmarks in the first category as the second category only has one implementation. However, the inter-accelerator choices are valid for both categories of the benchmarks as all of the graph algorithms have at least one multicore and a GPU version.

Multicore Benchmark Implementations: The unordered implementations of the benchmarks in the first category are acquired from the GAP benchmark suite [3], and the problem based benchmark suite (PBBS) [31] respectively. The unordered variants of Connected Components, BFS, DFS, PageRank, PageRank-DP, Coloring, Triangle Counting and Community are taken from CRONO [2]. The unordered variant for the A* workload is implemented in the CRONO suite. The KDG [18] and Galois [26] frameworks provide the relax-ordered and strict-ordered implementations of the benchmarks falling in first category, except A* that is implemented in these two frameworks. The destination node for A* is selected randomly while the heuristic is set to 0 as there is no additional information present with the input graphs.

GPU Benchmark Implementations: There are only relaxed and unordered implementations of a benchmark for the GPU. The Gunrock framework [34] provides the relax-ordered implementations for the benchmarks falling in the first category. The unordered implementation of SSSP, Graph Coloring, PageRank and PageRank-DP are taken from Pannotia [8], and the unordered version of BFS is obtained from Rodinia [9]. The unordered variants for the rest of the benchmarks are implemented using OpenCL within Pannotia.

Graph Inputs: Static input graphs utilized in this work are obtained from different domains, such as road networks, social networks, and biological networks. The static input graphs vary from extremely sparse (e.g. USA-CAL) to dense (e.g Twitter), and smaller in size (e.g Cage14) to large graphs (Friendster).

Table 3: Generated Temporal Graph and their characteristics. (s stands for static and t stands for temporal)

Graph (t)	Graph (s)	|V|	|E| (s)	deg (s)	|E| (t)	deg (t)
CAL_t_5	CAL	1.9M	4.7M	2.5	23.5M	12.4
CAL_t_10	CAL	1.9M	4.7M	2.5	47M	24.8
CAGE_t_5	CAGE	1.5M	25M	16.7	125M	83.3
CAGE_t_10	CAGE	1.5M	25M	16.7	250M	166.7
FB_t_5	Facebook	2.9M	41M	14.1	205M	71.7
FB_t_10	Facebook	2.9M	41M	14.1	410M	141.3
Orkt_t_5	Orkut	3M	234M	78	1.1B	390
Orkt_t_10	Orkut	3M	234M	78	2.3B	780
Twtr_t_5	Twitter	41M	1.4B	36	7.3B	179.26
Twtr_t_10	Twitter	41M	1.4B	36	14.7B	359

Different static input graphs along with their vertex count, edge count and average degree are shown in Table 3. These static input graphs are converted into temporal graphs through the graph generator described in Section 4. Table 3 also shows generated temporal graphs along with their vertex count, edge count and average degree. The value of T is set equal to 5 for t_5 graphs and 10 for t_10 graphs. t_5 graphs have 5 times more edges than static graph while t_10 graphs have 10 times more edges than static graphs. Vertices for all the graphs are divided into 2 equal parts (n) and time instants are also divided into 2 equal parts (n). Total number of probability distributions are 4 ($m * n$). The probability distributions used are unit Gaussian with scaling factors of 2, 4, 6 and 8.

The input graphs (static and temporal) that don't fit in the main memory of the accelerator are broken down into smaller chunks through the Stinger Framework [15], and processed sequentially. Memory transfer times are not included in the completion times for fair comparison between different accelerators. The final completion time of a benchmark-input combination doesn't contain the memory transfer times and only contains the execution time on the accelerator.

7 EVALUATION

This section presents the performance evaluation of different parallel implementations for various temporal graph benchmark and input combinations. The completion times for the temporal benchmark-input combinations are shown for Intel Xeon-40 core and GTX-1080 GPU machines, and compared with the completion times of their static counterparts. Additionally, this section shows that the choice of the optimal implementation for temporal graph benchmarks and inputs vary from their static counterparts.

7.1 Multicore Analysis

Table 4 shows the performance variations for all benchmark-input combinations (temporal and static) on the Intel Xeon 40-core machine. Unordered implementations of graph benchmarks, such as SSSP, BFS and Color that utilize static distribution of vertices/edges among cores are highly data parallel. These benchmarks along with large or dense static graphs (Orkut, Twitter and Friend) give

Table 4: Completion time (second) for static and temporal benchmark-input combinations on Intel Xeon 40-core. (SO) is strict-ordered, (RO) is relax-ordered and (UO) is unordered.

Graph	SSSP			BFS			Color			MST			A*			CC		
	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)
CAL	0.7	**0.57**	0.64	**0.69**	0.72	0.79	0.51	**0.15**	0.21	**0.56**	1.63	1.48	**0.42**	0.51	1.8	0.49	0.47	**0.17**
CAL_t_5	0.92	**0.48**	0.56	0.85	0.46	**0.48**	0.69	0.18	**0.17**	**0.65**	1.18	1.19	**0.59**	0.89	1.24	0.61	0.44	**0.19**
CAL_t_10	1.3	0.49	**0.38**	1.05	0.55	**0.49**	0.83	0.16	**0.12**	0.89	**0.75**	1.02	0.7	**0.6**	0.8	0.77	0.22	**0.15**
CAGE	0.9	**0.22**	0.44	0.88	**0.46**	0.63	0.74	**0.33**	0.39	**0.83**	1.16	2.9	**0.55**	0.69	0.72	0.61	0.16	**0.16**
CAGE_t_5	1.2	0.19	**0.1**	1.3	0.5	**0.45**	0.9	0.29	**0.21**	1.1	**1.05**	2.3	0.72	**0.53**	0.67	0.82	**0.12**	0.13
CAGE_t_10	1.7	0.34	**0.18**	1.9	0.61	**0.58**	1.2	0.36	**0.27**	1.05	**0.72**	1.6	0.91	**0.47**	0.77	0.8	0.07	**0.08**
FB	2.1	**0.32**	0.96	2.2	**1.02**	1.2	2.9	**1.57**	1.7	**2.1**	4.6	7.2	**1.9**	1.5	1.4	2.7	2.7	**2.3**
FB_t_5	3.2	1.14	**1.13**	3.8	1.45	**1.39**	3.2	1.2	**1**	**3.7**	2.8	1.2	3.4	**2.6**	2.7	2.9	**1.26**	1.26
FB_t_10	5.1	2.6	**2.5**	4.5	2.32	**2.25**	4.3	2.41	**2.3**	3.9	**2.1**	2.3	5.6	**2.63**	2.9	4.9	2.2	**2.18**
Orkt	6.7	3.78	**3.43**	6.4	**3.42**	3.45	5.9	2.91	**2.83**	**7.2**	7.02	7.3	**4.8**	7.9	7.8	6.2	2.67	**1.39**
Orkt_t_5	8.4	3.57	**4.23**	8.1	4.35	**4.3**	7.3	3.7	**3.1**	8.9	**4.6**	9.3	**5.5**	7.5	6.9	8.1	3.37	**2.9**
Orkt_t_10	10.2	4.1	**2.3**	10.8	2.4	**2.1**	9.4	4.3	**3.5**	2.3	**4.52**	5.74	7.6	**3.87**	9.2	11.4	4.1	**3.9**
Twtr	15.6	16.9	**14.2**	17.3	17.4	**9.7**	16.2	13.4	**12.9**	15.3	**5.4**	8.73	**13.7**	20.4	40.6	19.7	**10.1**	11.5
Twtr_t_5	18.3	16.9	**10.1**	23.2	12.4	**12.1**	19.5	13.8	**9.6**	17.5	**7.3**	9.4	**18.4**	21.1	58.2	26.3	8.26	**5.2**
Twtr_t_10	29.7	15.2	**12.8**	35.8	13.3	**11.9**	26.1	13.5	**12.9**	20.4	14.2	**13.1**	22.5	**19.3**	65.3	29.6	9.8	**6.7**

superior performance for unordered implementations. This is because they have large number of independent tasks that can be executed in parallel. In other words, the amount of unordered work in these benchmarks is high due to limited inter-task dependencies and low synchronization. Connected components with all input graphs give superior performance for unordered implementations for the same reason. Unordered implementations are also optimal for the temporal counterparts of the aforementioned benchmark-input combinations. However, temporal unordered versions show higher performance gains as compared to the static unordered versions for the same combinations. This is because they expose high parallelism as compared to their static versions.

On the other hand, graph benchmarks mentioned previously (SSSP, BFS and Color) running small and sparse temporal graphs (CAL, CAGE and FB) prefer relax-ordered implementations. Due to the small and sparse graphs, the amount of the work performed is relatively less, leading to limited parallelism. Unordered implementations in these cases perform high redundant work that ultimately hurts their performance. A task execution order is required for these combinations to reduce the amount of redundant work. Relax-ordered implementations satisfy this requirement by enforcing local order through per-core priority queues and hence extract superior performance. Additionally, the static distribution of the vertices/edges is not possible for these combinations as they don't have high vertex or edge level parallelism. In relax-ordered implementations, this is handled through sending tasks to different cores at runtime for dynamic load distribution. Alternatively, SSSP, BFS and Color with two temporal versions of small or sparse graphs (CAL_t_5, CAL_t_10, CAGE_t_5, CAGE_t_10, FB_t_5 and FB_t_10) prefer unordered implementations instead of relax-ordered. This is because the temporal versions of these graphs are no longer small or sparse. Instead, they have large number of temporal edges and high temporal density. This means that the temporal versions of these graphs have enough independent tasks that can be statically distributed among cores, and hence can be executed in parallel to extract superior performance. One notable exception is temporal SSSP-CAL_t_5 combination, which still gives better performance for relax-ordered implementation. This is because the diameter of the static and temporal CAL graph is relatively high as compared to other graphs. This means that there are longer dependency chains for the SSSP executing CAL graph (both static and temporal), which in turn requires the presence of the task execution order. Hence, relax-ordered implementation gives better performance for temporal SSSP-CAL combination.

Graph benchmarks that are not redundant work tolerant prefer strict-ordered implementations. This means that the redundant work is unable to expose parallelism in these workloads. They maintain global order at the cost of high synchronization to achieve the work efficiency of their sequential counterparts. Benchmarks such as A* with all input graphs prefer strict-ordered implementations. A* search finds a single path from a source to a destination vertex. Due to this, the amount of parallelism available in A* is limited. Thus, the performance of A* degrades if the implementation performs significant redundant work. MST with small and sparse graphs (CAL, CAGE and FB) also prefers strict-ordered implementations for the same reason. However, MST with large or dense graphs (Orkut and Twitter) prefers the relax-ordered implementation due to the presence of high reductions. Reductions in the implementation translate to work that is local to the core. Local work keeps the core busy due to which stringent ordering constraints can be relaxed. Strict-ordered implementations are an overkill for these combinations as they hinder parallelism. For temporal combinations, there is a shift in the choice of optimal implementation from strict-ordered towards relax-ordered for MST and A*. This shift is again created due to increase in the temporal edges or temporal density of the input graph, which in turn allows the inner loop in the relax-ordered implementation to expose more parallelism.

Table 5: Completion time (second) for static and temporal benchmark-input combinations on NVidia GTX-1080 GPU. (SO) is strict-ordered, (RO) is relax-ordered and (UO) is unordered.

Graph	SSSP			BFS			Color			MST			A*			CC		
	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)
CAL	-	**1.3**	2.8	-	**1.1**	1.15	-	**1.4**	1.9	-	**1.7**	2.1	-	**1.5**	3	-	1.1	**0.8**
CAL_t_5	-	**2.1**	2.3	-	1.9	**1.1**	-	**1.6**	2.3	-	2.4	**2.2**	-	**2.6**	3.3	-	1.85	**1.32**
CAL_t_10	-	4.5	**2.7**	-	4.1	**2.3**	-	3.6	**2.5**	-	3.8	**2.7**	-	4.3	**3.9**	-	3.8	**2.1**
CAGE	-	3.6	**2.9**	-	**3.2**	3.7	-	**4.1**	5.2	-	**3.1**	5.9	-	**5.7**	8.3	-	4.6	**3.1**
CAGE_t_5	-	4.8	**3.2**	-	5.1	**3.9**	-	6.7	**5.3**	-	**4.8**	6.2	-	**6.9**	7.4	-	4.8	**3.6**
CAGE_t_10	-	5.1	**3.4**	-	7.2	**3.6**	-	6.9	**5.7**	-	5.8	**5.7**	-	**8.3**	9.2	-	5.9	**4.2**
FB	-	**4.3**	8.7	-	**5.1**	6.1	-	**4.9**	5.6	-	**5.8**	7.5	-	**7.9**	10.3	-	6.2	**5.7**
FB_t_5	-	**5.9**	9.2	-	6.7	**5.2**	-	8.2	**5.5**	-	9.5	**7.4**	-	**8.8**	11.4	-	13.9	**6.4**
FB_t_10	-	10.8	**9.8**	-	8.4	**7.3**	-	11.5	**6.2**	-	13.8	**8.1**	-	14.5	**13.1**	-	14.5	**6.1**
Orkt	-	6.5	**6.1**	-	**7.4**	7.6	-	**6.2**	6.9	-	**8.7**	12.5	-	**6.1**	10.2	-	5.9	**5.1**
Orkt_t_5	-	7.9	**5.9**	-	8.6	**8.4**	-	7.7	**7.1**	-	**10.1**	14.7	-	9.3	12.5	-	7.3	**4.9**
Orkt_t_10	-	9.6	**6.1**	-	10.3	**9.7**	-	9.1	**8.6**	-	**12.3**	15.1	-	**11.6**	14.3	-	10.5	**7.4**
Twtr	-	10.2	**8.7**	-	11.4	**9.9**	-	10.5	**8.7**	-	**12.5**	20.4	-	14.2	18.1	-	11.2	**10.5**
Twtr_t_5	-	12.5	**9.7**	-	14.7	**10.8**	-	13.7	**9.1**	-	**16.4**	17.1	-	20.3	**17.7**	-	15.3	**11.1**
Twtr_t_10	-	15.8	**10.3**	-	17.4	**11.6**	-	20.1	**12.5**	-	21.6	**18.2**	-	23.6	**19.4**	-	17.8	**12.3**

The geometric mean completion time for the optimal implementation of all benchmark-input combinations is calculated using the static choices for temporal combinations, and then utilizing true temporal choices for temporal graphs. These choices are highlighted in Table 4. True temporal choices give a performance gain of 25% over static choices for t_5 (graphs with 5 times more temporal edges than static) graphs, and 40% over static choices for t_10 (graphs with 10 times more temporal edges than static) graphs on Intel Xeon-40 machine. This shows that the optimal implementation choices for static graph benchmarks and inputs cannot be applied directly for temporal graph benchmarks and inputs. Hence, temporal graphs should be considered independently when deciding their optimal parallel implementation on a multicore machine.

7.2 GPU Analysis

Table 5 shows the completion times for all benchmark-inputs combinations (temporal and static) on GTX-1080 GPU. The GPU trends are similar to multicore except that all benchmark-input combinations that give superior performance for strict-ordered on multicore, now map best to relax-ordered on the GPU.

Static graph benchmarks with high vertex or pareto level parallelism (such as SSSP, BFS and Color) along with large or dense graphs (Orkut and Twitter) give optimal performance for unordered implementations. This is due to the presence of large number of independent tasks that can be executed in parallel. Extremely data parallel benchmarks, such as Connected Component in which task execution order is not required at all, also choose unordered implementations as optimal for all input graphs. Alternatively, SSSP, BFS and Color running small or sparse graph give superior performance for relax-ordered implementations due to the presence of local per-core order. Additionally, benchmarks that are limited redundant work tolerant, such as A* with all input graphs, and MST with small and sparse graphs also prefer relax-ordered implementations as they don't have enough independent tasks to execute in parallel. Observing the completion times for temporal combinations reveal that for SSSP, A*, BFS, Color and MST there is a shift in the choice of the optimal implementation from relax-ordered implementation (for static graphs) to unordered implementation (for temporal graphs). This shift in the choice of optimal implementation is again due to the high temporal density and edges.

As in case of multicore, the geometric mean completion time of the optimal implementation for all benchmark-input combinations are calculated using the static choices for temporal combinations and then utilizing true temporal choices for temporal graphs on GTX-1080 GPU. The true temporal choices give a performance gain of 20% over static choices for Temporal_5 (graphs with 5 times more temporal edges than static) graphs, and 35% over static choices for Temporal_10 (graphs with 10 times more temporal edges than static) graphs on GTX-1080 GPU. Again, these results suggest that temporal graphs should be considered independently when deciding their optimal parallel implementation on a GPU machine.

7.3 Multicore and GPU Comparisons

Figure 4 shows the performance comparison of Intel Xeon 40-core machine with NVidia GTX-1080 for different benchmarks. Each benchmark has three completion times for both GPU and multicore, namely static (averaged over all static graphs), temporal_5 (average over all temporal graphs with 5 times more edges than static graphs), and temporal_10 (average over all temporal graphs with 5 times more edges than static graphs). All completion times in the figure are normalized to their respective GPU's completion times. From static to temporal_10, there is a change in the accelerator that gives superior performance, i.e., the optimal accelerator changes from multicore to GPU. The reason for this is that as we move towards temporal_10, there is an increase in temporal density, which in turn exposes higher parallelism. This parallelism is exploited better by the GPU due to the presence of large number of threads. Even

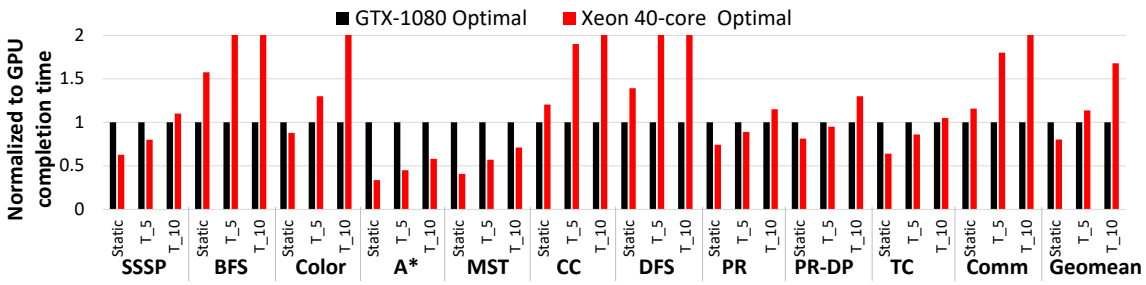

Figure 4: Performance comparison of Intel Xeon 40-core and GTX-1080 GPU for all graph benchmarks with average across all static graphs, all temporal_5 graphs and all temporal_10 graphs. Completion Time is normalized w.r.t. GPU completion time (Higher is worse)

for benchmarks, such as A* and MST that don't shift to GPU for temporal graphs, the results show better performance on GPU as compared to their static counterparts.

The static graphs show a geometric mean performance gains of 24.5% on Intel Xeon 40-core over GTX-1080 GPU. However, the temporal_5 and temporal_10 graphs improve performance by 13.74% and 67.8% respectively for the GPU over the Intel Xeon multicore. Hence, the choice of an accelerator also shifts for static versus temporal graphs.

8 CONCLUSION

This paper proposes a graph generator method to transform a static graph into temporal graph by applying various probability distributions to different parts of the static graph. After generating the temporal graph, this work explores the parallel implementation choices for diverse temporal graph benchmarks to extract the optimal performance on different parallel machines. The choice of the optimal parallel implementation leads to a geometric performance gain of 46.38% on Intel Xeon 40-core, and 20.30% on the NVidia GTX-1080 for the generated temporal graphs. This work also highlights that the optimal implementation for temporal graphs and their static counterparts are not always the same.

ACKNOWLEDGMENTS

This work was funded by the U.S. Government under a grant by the Naval Research Laboratory. This work was also supported by the National Science Foundation (NSF) under Grant No. CNS-1718481.

REFERENCES

[1] Masab Ahmad, Halit Dogan, Christopher J Michael, and Omer Khan. 2019. HeteroMap: A Runtime Performance Predictor for Efficient Processing of Graph Analytics on Heterogeneous Multi-Accelerators. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 268–281.

[2] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *2015 IEEE International Symposium on Workload Characterization*. 44–55. https://doi.org/10.1109/IISWC.2015.11

[3] Scott Beamer, Krste Asanovic, and David Patterson. 2015. Locality Exists in Graph Processing: Workload Characterization on an Ivy Bridge Server. In *Int. Symposium on Workload Characterization (IISWC)*.

[4] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. 2016. End to End Learning for Self-Driving Cars. *CoRR* abs/1604.07316 (2016).

[5] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2019. ChronoGraph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering* (2019).

[6] Vince D Calhoun, Robyn Miller, Godfrey Pearlson, and Tulay Adalı. 2014. The chronnectome: time-varying connectivity networks as the next frontier in fMRI data discovery. *Neuron* 84, 2 (2014), 262–274.

[7] Arnaud Casteigts, Paola Flocchini, Walter Quattrociocchi, and Nicola Santoro. 2011. Time-varying graphs and dynamic networks. In *International Conference on Ad-Hoc Networks and Wireless*. Springer, 346–359.

[8] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 185–195. https://doi.org/10.1109/IISWC.2013.6704684

[9] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *2009 IEEE International Symposium on Workload Characterization (IISWC)*. 44–54. https://doi.org/10.1109/IISWC.2009.5306797

[10] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). 2009. *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Vol. 74. DIMACS/AMS.

[11] Ugur Demiryurek, Farnoush Banaei-Kashani, and Cyrus Shahabi. 2010. A case for time-dependent shortest path computation in spatial networks. In *Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. ACM, 474–477.

[12] Dingxiong Deng, Cyrus Shahabi, Ugur Demiryurek, Linhong Zhu, Rose Yu, and Yan Liu. 2016. Latent space model for road networks to predict time-varying traffic. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 1525–1534.

[13] Bolin Ding, Jeffrey Xu Yu, and Lu Qin. 2008. Finding time-dependent shortest paths over large graphs. In *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, 205–216.

[14] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *The 36th ACM Conference on Programming Language Design and Implementation (PLDI)*. ACM, New York, NY, USA, 12.

[15] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680

[16] Betsy George and Shashi Shekhar. 2008. Time-aggregated graphs for modeling spatio-temporal networks. In *Journal on Data Semantics XI*. Springer, 191–212.

[17] Wentao Han, Youshan Miao, Kaiwei Li, Ming Wu, Fan Yang, Lidong Zhou, Vijayan Prabhakaran, Wenguang Chen, and Enhong Chen. 2014. Chronos: a graph engine for temporal graph analysis. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 1.

[18] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. 2015. Kinetic Dependence Graphs. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 457–471. https://doi.org/10.1145/2694344.2694363

[19] Jialin He and Duanbing Chen. 2015. A fast algorithm for community detection in temporal network. *Physica A: Statistical Mechanics and its Applications* 429 (2015), 87–94.

[20] Weishu Hu, Haitao Zou, and Zhiguo Gong. 2015. Temporal pagerank on social networks. In *International Conference on Web Information Systems Engineering*. Springer, 262–276.

[21] Silu Huang, James Cheng, and Huanhuan Wu. 2014. Temporal graph traversals: Definitions, algorithms, and applications. *arXiv preprint arXiv:1401.1919* (2014).

[22] Silu Huang, Ada Wai-Chee Fu, and Ruifeng Liu. 2015. Minimum spanning trees in temporal graphs. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 419–430.

[23] Klaus Kofler, Ivan Grasso, Biagio Cosenza, and Thomas Fahringer. 2013. An Automatic Input-sensitive Approach for Heterogeneous Task Partitioning. In *Proceedings of the Int. Conference on Supercomputing (ICS '13)*. ACM, New York, NY, USA, 149–160.

[24] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web*. ACM, New York, NY, USA, 591–600. https://doi.org/10.1145/1772690.1772751

[25] Shouwen Lai and Binoy Ravindran. 2010. On distributed time-dependent shortest paths over duty-cycled wireless sensor networks. In *2010 Proceedings IEEE INFOCOM*. IEEE, 1–9.

[26] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 456–471. https://doi.org/10.1145/2517349.2522739

[27] Vincenzo Nicosia, John Tang, Mirco Musolesi, Giovanni Russo, Cecilia Mascolo, and Vito Latora. 2012. Components in time-varying graphs. *Chaos: An interdisciplinary journal of nonlinear science* 22, 2 (2012), 023101.

[28] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*. http://networkrepository.com

[29] Polina Rozenshtein and Aristides Gionis. 2016. Temporal pagerank. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 674–689.

[30] Nicola Santoro, Walter Quattrociocchi, Paola Flocchini, Arnaud Casteigts, and Frederic Amblard. 2011. Time-varying graphs and social network analysis: Temporal indicators and metrics. *arXiv preprint arXiv:1102.0629* (2011).

[31] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *SPAA*.

[32] John Tang, Salvatore Scellato, Mirco Musolesi, Cecilia Mascolo, and Vito Latora. 2010. Small-world behavior in time-varying graphs. *Physical Review E* 81, 5 (2010), 055101.

[33] Scott Le Vine, Alireza Zolfaghari, and John Polak. 2015. Autonomous cars: The tension between occupant experience and intersection capacity. *Transportation Research Part C: Emerging Technologies* 52 (2015), 1 – 14. https://doi.org/10.1016/j.trc.2015.01.002

[34] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 49 pages. https://doi.org/10.1145/3108140

[35] Yishu Wang, Ye Yuan, Yuliang Ma, and Guoren Wang. 2019. Time-Dependent Graphs: Definitions, Applications, and Algorithms. *Data Science and Engineering* (2019), 1–15.

[36] Klaus Wehmuth, Artur Ziviani, and Eric Fleury. 2015. A unifying model for representing time-varying graphs. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 1–10.

[37] Huanhuan Wu, James Cheng, Silu Huang, Yiping Ke, Yi Lu, and Yanyan Xu. 2014. Path problems in temporal graphs. *Proceedings of the VLDB Endowment* 7, 9 (2014), 721–732.

[38] Huanhuan Wu, James Cheng, Yiping Ke, Silu Huang, Yuzhen Huang, and Hejun Wu. 2016. Efficient algorithms for temporal path computation. *IEEE Transactions on Knowledge and Data Engineering* 28, 11 (2016), 2927–2942.