Exploring Parallel Algorithmic Choices for Graph Analytics

Akif Rehman
akif.rehman@uconn.edu

Exploring Parallel Algorithmic Choices for Graph Analytics

Akif Rehman

BSc. Electrical Engineering,

University of Engineering & Technology,

Lahore, Pakistan,

2009

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Master of Science

at the

University of Connecticut

2019

APPROVAL PAGE

Master of Science Thesis

Exploring Parallel Algorithmic Choices for Graph Analytics

Presented by

Akif Rehman, B. Sc.,

Major Advisor	Omer Khan
Associate Advisor	Lei Wang
Associate Advisor	John Chandy

University of Connecticut

2019

ACKNOWLEDGMENTS

I would like to thank my parents Mr and Mrs Waheed-ur-Rehman for providing me the best possible education and making me capable for carrying out this work. I would also like to thank my wife Sheeza Anwar Shah for standing by me in the most difficult times and for being there when I needed her the most. I hope she always finds happiness and never sees another moment of sadness.

I would also like to thank my advisor Omer Khan for guiding me through my Masters journey. This work would not have been possible without his valuable insights

I would especially like to thank my friend Hamza Omar for his support throughout my stay at the University of Connecticut. I would not be able to make this far without him. He is the best person I have met in my entire lifetime. At the end I would like to thank all of my friends Masab, Mohsin, Usman, Rizwan, Sohaib, Tara and Sumaira for their support.

This Masters thesis is the end of an era for me.

Contents

List of Figures

List of Tables

Exploring Parallel Algorithmic Choices for Graph Analytics

Akif Rehman,

University of Connecticut, 2019

ABSTRACT

Sequential graph algorithms are implemented through ordered execution of tasks to achieve high work efficiency. Exposing parallelism in these ordered workloads tends to be an elusive problem. Strict-ordered parallel implementations find nodes that don't have read-write dependencies and hence can be executed in parallel. They have the work efficiency of their sequential counter-parts due to strict ordering constraints. Larger amount of parallelism can be achieved at the expense of redundant work. Relax-ordered implementations remove the global order and only impose the local order. They go through multiple iterations and have the property of monotonically increasing or decreasing output values allowing them to converge efficiently. Unordered implementations move one step ahead and remove the local order as well. Due to the absence of the order, a large amount of redundant work is done but at the same time more parallelism is exposed. Different parallel implementations perform optimally for different algorithms. Similarly, as the graph input changes, the optimal parallel version may change. The choice of optimal parallel implementations is strongly correlated with the characteristics of graph benchmark and input. This work proposes an analytical prediction model that chooses the optimal parallel implementation for a given benchmark-input combination on a single accelerator setup e.g. multicore or GPU. The prediction model is also integrated with a state-of-the-art performance predictor that lacks this capability on a multi-accelerator setup. The proposed predictor shows geometric performance gains of 54% on a multicore, 14% on a GPU, and 31.5% in a multi-accelerator setup. However, many real-world systems utilize graphs that are time-varying in nature. In time-varying graphs edges appear and disappear with respect to time. Additionally, the weights of different edges are also a function of time. Various conventional graph algorithms such as SSSP have been developed for time-varying graphs. However, these algorithms are sequential in nature and their parallel counter-parts are largely overlooked. Alternatively, parallel algorithms for static graphs are implemented as ordered and unordered variants. The parallel implementations are also adopted for temporal graphs to explore the implementations that provide optimal performance on different accelerators. This work also 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 and the geometric performance gain of 20% can be achieved by selecting the ideal optimal choices for temporal graphs instead of the choices for static graphs.

1 PERFORMANCE PREDICTOR FOR STATIC GRAPHS

1.1 MOTIVATION

Algorithms involving graph workloads [1] are widely utilized due to their omnipresent applications [2]. They are able to find their way in different domains such as self-driving cars [3], social network analytics, traffic map applications [4] and routing algorithms [5]. A graph algorithm executes tasks that read a vertex and its corresponding edges, perform computations on them, and write the output into some global data structure [6]. Sequential implementations of these algorithms are highly work efficient as they employ queueing primitives to maintain a global order. There is a need for ordering in these workloads due to read-write dependencies on prior and future tasks with parent-child relationships between the tasks.

In order to unlock parallelism in these workloads, previous works [7] have introduced their unordered counterparts e.g, Bellman Ford [8] for the single source shortest path problem. The unordered versions remove ordering constraints as different tasks execute in any order and can thus be executed in parallel. However, this parallelism is achieved at the expense of redundant work, which is required for convergence in unordered versions. Due to the inferior work efficiency of unordered workloads, prior works such as KDG [9] have devised frameworks exposing parallelism in ordered implementations. These works use task level speculation to execute future tasks with strict ordering guarantees. KDG applies safe-source tests via synchronization on tasks before they are executed concurrently to enforce ordering constraints due to read-write task dependecies. This results in the same work-efficiency as is availabe

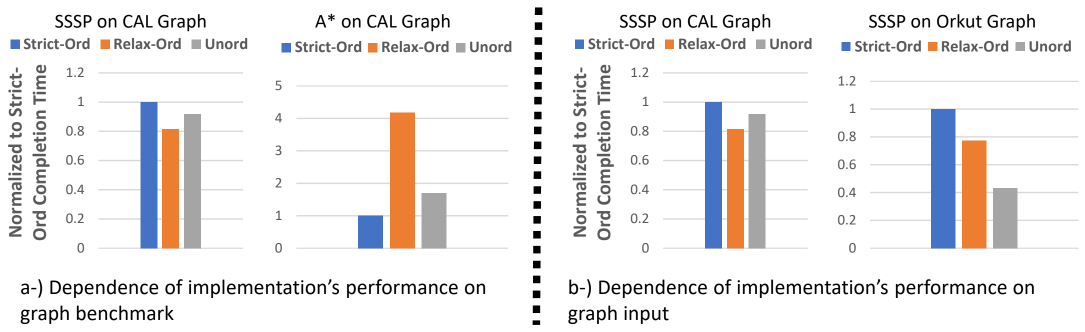

Figure 1.1: How graph workload and input variations exhibit different performance across strict-ordered, relax-ordered and unordered implementations

in sequential versions. However, this redundant work is pruned out at the cost of augmented synchronization. In order to strike a balance between work-efficiency and parallelism, the Galois framework [10, 11] relaxes the ordering constraints. This is achieved by removing the global order and by maintaining a local order among tasks through per-core priority queues.

Looking at the prior literature, parallel variants of graph workloads can be divided into three categories: unordered, relax-ordered and strict-ordered. Going from unordered towards strict-ordered implementations, work efficiency becomes better and synchronization increases. This categorization of graph workloads poses a question. Does one parallel variant always have superior performance for all graph workloads or is the choice of the parallel implementation workload dependent? In order to put this in perspective, let's consider two workloads that are quite similar to each other, SSSP and A*. SSSP finds the shortest path from a source node to all other nodes in a graph. A* finds the shortest path from a source node to a destination node in a given graph. Figure 1.1a shows the normalized completion times for OpenTuner optimized [12] SSSP and A* for strict, relax and unordered parallel variations running on the Intel Xeon E5-2650 v3 (40 core) machine for California Road Network. It can be seen that for SSSP, the relax order variant gives the best completion time. While for A*, the strict-ordered variant performs the best. The reason for this is that there is plenty of parallelism available in SSSP as there are a lot of independent paths. In this case, the strict ordered version hinders parallelism as synchronization impacts performance. In contrast to this, there is not much parallelism available in A* as we need to find a single path. For A*, the strict ordered version performs the best as it performs no redundant work compared to the other relaxed-ordered and unordered versions. In this case, synchronization is not hurting performance because the work that needs to be done is highly ordered. This shows that variations in workloads can lead to different choices.

Input dependence [13] also plays a significant role in graph workload's performance, and can lead to a different choice. Figure 1.1b shows the normalized completion times for OpenTuner optimized SSSP for the three parallel variants running on the Intel Xeon 40 core machine for sparse (California Road Network) and dense (Orkut) graphs. It is clear from the figure that the relax-ordered variant gives the best performance for the CAL graph, while the unordered variant performs optimally for the Orkut input. Due to having a higher graph diameter, longer dependency chains in the California road network make it suitable for the relax-ordered variant as maintaining the order results in less redundant work. The ordering constraints in the case of the Orkut graph are not as stringent as they were in case of CAL graph due to low diameter, hence the unordered version exhibits a superior performance.

The correlation of ordered or unordered choices with benchmark-input combinations poses two questions. What patterns of benchmark-input combinations translate to the choice of a specific implementation on a multi-core or a GPU? Which combinations of benchmark-input-implementation lead to the choice of a particular accelerator? This motivates the need to tune the implementation-accelerator search space for various benchmark-input combinations. There is a need to design an inter-implementation choice model that can predict the optimal implementation for a benchmark-input combination. Prior works such as OpenTuner [12] and PetaBricks [13, 14, 15] have optimized algorithmic choices for standalone implementations. Their work is not able to choose the best implementation. HeteroMap [16] is a state-of-the-art performance predictor that selects the accelerator and optimizes the intra-accelerator parameters for unordered graph workloads. Intra-accelerator choices are various concurrency parameters that HeteroMap exposes e.g. optimal core count, multithreading, SIMD and thread placement mechanism for multicore, and local and global threading for GPU. HeteroMap assumes that there is just one implementation(unordered) running on CPU and GPU and it picks either the GPU-unordered or CPU-unordered. HeteroMap is unable to differentiate between strict-ordered, relax-ordered and unordered implementation for a graph problem. Performance of the HeteroMap framework can significantly improve by integrating an inter-implementation predictor with it.

This work proposes an inter-implementation predictor that creates a mapping to choose the right algorithm from graph benchmarks and inputs for various parallel accelerator setups, such as multicore or a GPU. The mechanism of quantitative representation of graph benchmarks and inputs are taken from the HeteroMap framework. An analytical decision tree model is created that takes benchmark-input combination and predicts the optimal implementation as an output. The proposed predictor is also integrated with the HeteroMap framework to extract superior performance. Given a graph problem and an input graph, the proposed predictor optimizes the inter-implementation choice. The performance enhancement takeaways of this work are outlined below:

- The proposed predictor selects the optimal parallel implementation for the given benchmark-input combination on multicore. This leads to the geometric performance gains of 70%, 40% and 55% over selecting strict-ordered, relax-ordered and unordered implementations respectively for all of the benchmark-input combinations on the Intel Xeon E5-2650 v3 multicore machine.
- The proposed predictor selects the optimal parallel implementation for the given benchmark-input combination on GPU. This leads to the geometric performance gains of 5% and 39% over selecting relax-ordered and unordered implementations respectively for all of the benchmark-input combinations on the NVidia GTX-1080 GPU.
- The proposed predictor is also integrated with the HeteroMap framework. This gives a geometric gain of 32% over original HeteroMap's performance because now it can select the optimal implementation with the optimal accelerator.

1.2 ALGORITHMIC-CENTRIC CLASSIFICATION OF GRAPH WORKLOADS

Task-level parallelism makes parallel programming effortless, and has seen a rise in traction as different concurrent-programming frameworks [9, 10] have seamlessly integrated it. From a programmer's perspective, a task needs to be defined that performs some computation and runs in parallel with other tasks, hence exposing parallelism. Low-level system details such as thread synchronization and load balancing are managed through the constructs provided by the framework or library. Due to the fact

Strict-ordered

```
1. Queue taskQueue (Local)
   List orderList (Global)
2. For each task in taskQueue do:
3.  task = taskQueue.peek()
4.  test = safe_source_test(task)
5.  if test == pass
6.   task = taskQueue.pop()
7.   Atomic:
     remove_entry(orderList(task))
8.   For each child of task do:
9.    task = taskQueue.push()
      OR send_to_remote_core()
10.   critical_section(sharedData)
11.   Atomic:
      add_entry(orderList(child))
```

Relax-ordered

```
1. Queue taskQueue (Local)
2. For each task in taskQueue do:
3.  task = taskQueue.peek()
4.  test = local_test(task)
5.  if test == pass
6.   task = taskQueue.pop()
7.   For each child of task do:
8.    task = taskQueue.push()
      OR send_to_remote_core()
9.    critical_section(sharedData)
```

Unordered

```
1. List taskList (Local)
2. For each task in taskList do:
3.  task = taskList.peek()
4.  task = taskList.pop()
5.   For each child of task do:
6.    task = taskList.push()
7.    critical_section(sharedData)
```

Figure 1.2: Generic Pseudocode of Strict-Ordered, Relax-Ordered and Unordered Implementations

that this execution model is machine-independent, task parallel algorithms acquire the advantage of scalability at higher core-counts. All task parallel algorithms follow some kind of task ordering execution model to some extent. It might be possible that the execution of a specific task is dependent on the execution of some other tasks or thread-level synchronization is required by a set of tasks so that their inter-task dependencies are forwarded properly. There may also be some shared data with read-write dependencies that is being modified by some tasks. Due to the inter- and intra-task dependencies, thread synchronization becomes a vital component of the execution model. In an ideal task parallel algorithm, all tasks are executed in parallel and there are no inter or intra dependencies.

In strict-ordered task parallel algorithms, a strict order is enforced on the execution of tasks distributed among cores. Work efficiency in these implementations is same as their sequential counterparts. Enforcing a global order on tasks execution and trying to extract parallelism among tasks at the same time is not an easy problem. Kinetic dependence graph(KDG) [9] is one of concurrent programming frameworks that supports task-level parallelism and enforces a strict order on tasks. In KDG there is a local priority queue per core and a global order is enforced by ordering task insertions and deletions through an ordered list shared among all the cores. Every time a task is dequeued from the local priority queue, a safe-source test is executed that checks whether the current task is dependent on the execution in other cores. If the dependencies are found, then the dequeue operation waits till that dependency is resolved. If there are no dependencies among tasks, they can execute in parallel in their respective cores. The key idea is to find an independent set of tasks equal to the number of cores so that they can execute concurrently. Figure 1.2 shows the generic pseudocode of a strict-ordered graph benchmark. Benchmark implementation defines a priority queue for each core and a global ordered list shared among cores. Each core peeks a task from its local priority queue and runs safe-source test on that task to verify that there is no such task in any other core. If the safe-source test passes, then the task is dequeued from the priority queue and atomically removed from the shared ordered list. After the dequeue operation, each child is enqueued to either the local priority queue or send as a message to the priority queue of some other core after some shared data operation. Each child is also atomically added to the ordered list for future safe-source tests.

Strict-ordered algorithms hinder parallelism due to the constraint of global order that needs to be followed. In contrast to this more parallelism can be exposed at the expense of more redundant work. The Galois [10] framework, for example, adopts this approach by having a local priority queue per core while ignoring the strict global task execution order enforced by KDG. In this kind of implementation, thread synchronization is reduced as only the intra-task dependencies now require synchronization instead of both inter and intra task dependencies. However, relax-ordered algorithms go through multiple iterations before they converge hence the amount of redundant work is higher as compare to strict-ordered. Figure 1.2 shows the generic pseudocode of a relaxed-ordered graph benchmark. It has a local priority queue for each core but the ordered list like the one in strict-ordered is removed as the global order is not required. Instead of running the safe-source test on the task through shared ordered list, a local test is executed on the task. If the test passes, each child of the task is either enqueued to the local priority queue or send as a message to some other core for enqueueing after the critical section.

Unordered task parallel algorithms do not enforce local or global order. They are structured in a way that there are no inter dependencies between the tasks and a task can be executed in parallel with other tasks. As there is no order (local or global), unordered algorithms may go through a lot of iterations before convergence. The amount of redundant work in unordered algorithms is usually high due to the large number of iterations. Work efficient implementation of unordered algorithms scale at higher core counts and provide better performance than their relax-ordered counterparts. There is still some synchronization present in unordered algorithms due to read-write shared data dependencies. In the absence of shared data, an unordered algorithm implementation has different phases and thread-level synchronization is done between phases through barriers. This ensures that all the dependencies are propagated properly and the threads are ready to execute next phase of the algorithm. Figure 1.2 shows the generic pseudocode of unordered graph benchmarks. The benchmark does not define a priority queue instead it specifies a list as the tasks can be executed in any order. When a core peeks a task from its local list, no additional tests are performed on the task due to the absence of task execution order. Each child of the task is added to the local list after passing it through critical section.

The pseudocodes of different implementations reveal that there are following trade-offs between parallelization strategy, synchronization and work efficiency of task-parallel workloads:

- In the absence of any order(local or global), the parallelization strategy is to statically distribute the vertices/edges among cores. This exposes higher parallelism but at the same time performs higher redundant work.
- Enforcing a local order through per core queues and dynamically distributing the vertices/edges among cores perform less redundant work with exposing reasonable parallelism.
- Global order is enforced at the expense of higher synchronization that leads to optimal work efficiency but at the same time significantly reduces the exposed parallelism.

It is evident from the discussion above that each parallel implementation of the graph benchmark has its own set of characteristics. The variations in the characteristics of different implementations are driving forces that can lead to the selection of the optimal implementation. The next section proposes a framework that creates a decision tree forthe optimal implementation selection based on the high-level characteristics, convert these high-level characteristics into discrete variables and then formulates the decision tree based on these discrete variables.

1.3 PROPOSED FRAMEWORK FOR IMPLEMENTATION SELECTION

This section proposes a framework that has the capability of selecting the optimal implementation for a given graph benchmark and input[11]. We first construct a decision tree for the optimal implementation selection based on the high level characteristics of the implementation such as parallelization strategy, ordering constraints, synchronization and redundant work and the characteristics of the graph input such as size and density. The benchmark (B) and input variables (I) are then introduced that capture these high level characteristics through quantitative representations. The B and I variables provide a mechanism for programmers to represent different implementations of a benchmark and input graphs in the form of discrete numbers. These representations are either achieved by visual inspection or benchmarking, making it simpler for the programmers. Once the B and I variables are defined, the inter-implementation decision tree based on the high level characteristics is translated into a decision tree that utilizes the B and I variables. The workflow of the inter-implementation decision tree (based on B and I variables) is also demonstrated through an example on a multicore and a GPU for the specific graph benchmarks and inputs.

High Level Inter-Implementation Selection Model

The decision tree using characteristics of different implementations of a benchmark and graph input is proposed in this section. Algorithm 1 shows the pseudocode for the inter-implementation selection decision tree model. The inputs of this decision tree

[11]1. The intra machine concurrency parameters for a benchmark implementation are optimized through HeteroMap [16] (c.f. Section 1.4)

Algorithm 1 Inter-Implementation Decision Tree using high-level implementation characteristics and input graph

```
Input: Strict-Ordered (SO), Relax-Ordered (RO), Unordered (UO) Implementations and Input Graph (G)
Output: Optimal Implementation
 1: procedure DECISION TREE($UO$, $RO$, $RO$, $G$) return $Optimal\ Implementation$
 2:     $Optimal\ Implementation$ = ""
 3:     if High unordered work in $UO$ then
 4:         if Low Synchronization in $UO$ then
 5:             if $G$ is large or dense then
 6:                 $Optimal\ Implementation$ = Unordered
 7:     if $UO$ not selected then
 8:         if Less unordered work in $RO$ then
 9:             if Presence of task execution order then
10:                 if Low Synchronization in $RO$ then
11:                     if Less Reductions in $RO$ then
12:                         if $G$ is small or sparse then
13:                             $Optimal\ Implementation$ = Relax-ordered
14:                     else if High Reductions in $RO$ then
15:                         if $G$ is large or dense then
16:                             $Optimal\ Implementation$ = Relax-ordered
17:         if $RO$ not selected then
18:             $Optimal\ Implementation$ = Strict-Ordered
```

are three implementations (unordered, relax-ordered and strict-ordered) for a graph benchmark, and the input graph while the output is the selection of the optimal implementation (*Line* 1). In the begining the optimal implementation parameter is empty indicating that no implementation is selected yet.

An unordered implementation is selected when all of the following conditions are satisfied (*Line* $3 - 6$):

- Tasks in the implementation are such that they can be executed independent of each other i-e the amount of unordered work is significant. This is because the unordered work indicates the presence of high parallelism in the implementation. This means that the read-write dependencies between the tasks are minimal and hence they can be executed in parallel. High unordered work also signifies that the load can be statically distributed among the cores and there is no need for the presence of the ordering constraints.

- Synchronization in the unordered implementation is low. Unordered algorithms are iterative in nature and the amount of redundant work performed by them is strongly correlated with the number of iterations. More iterations translate to high redundant work. When the synchronization is high it leads to missed read-write dependencies, hence increasing the number of iterations and consequently the amount of redundant work. Unordered implementations already execute higher number of tasks as compared to their relax-ordered or strict-ordered counter-parts to expose parallelism. Higher synchronization further increases the amount of redundant work, thus making the unordered implementation suboptimal.

- The size of the input graph is large or it is a dense graph. This is becaue large number of vertices or edges translate to the larger number of independent tasks that an unordered implementation may execute in parallel. An unordered implementation in this case has enough non-redundant work that is contributing to its convergence. On the other hand if the graph is small or sparse, there won't be enough opportunites for parallelism in the input graph. This may cause unordered implementations to perform significant amount of redundant work which in turn indicates that a task execution order is required. Hence the unordered implementations are relatively suitable for larger or dense graphs. They can also be selected in case of smaller or sparse graphs if the benchmark is completely data parallel with negligible data dependencies.

If the unordered implementation is not selected ($Line\ 7$), the decision tree inspects the characteristics of the relax-ordered implementation along with the input graph. A relax-ordered implementation is selected when all of the following conditions are satisfied ($Line\ 8-16$):

- Number of tasks to be executed by the implementation are not high, making them sensitive to the read-write dependencies. This means that there are not enough independent tasks that can be executed in parallel with each other. Due to this the static distribution of the load among cores is unable to extract parallelism. A task execution order is required so that these read-write dependencies can be captured effectively. In the relax-ordered implementations, a local order is enforced on the task execution through per core priority queues. The presence of ordering primitives in the relax-ordered variants make them optimal for such scenarios.

- Synchronization in the relax-ordered implementation is low. Relax-ordered implementations have monotonically increasing or decreasing output values that help them converge. High synchronization results in missed read-write dependencies causing the relax-ordered versions to go through more iterations. Higher the number of iterations, larger the amount of redundant work, hence the synchronization should be low for the selection of the relax-ordered implementations.

- The size of the input graph is small or it is a sparse graph with negligible reductions in the implementation. This shows that the number of tasks to be executed are less. Lesser the amount of available work, higher the need for a task execution order. As the relax-ordered implementations enforce the local order,

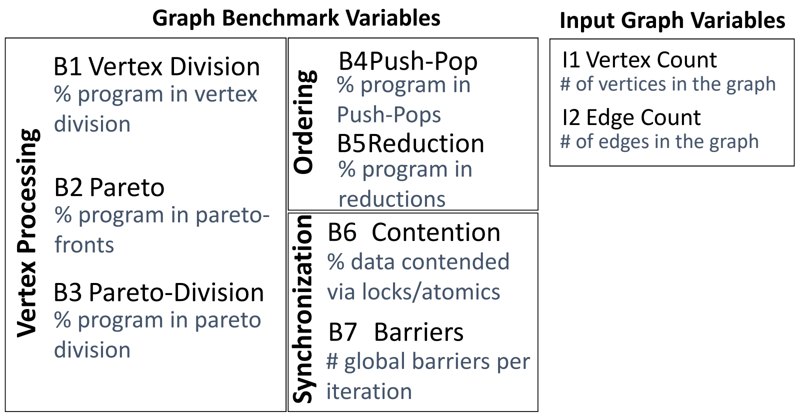

Figure 1.3: Graph Benchmark B and Input I variables. Each implementation of the graph benchmark will be expressed in B variables and an input graph will be represented in I variables

they are the optimal choice. Relax-ordered implementations can also be selected for larger and dense graphs when the implementation has high reductions. This is because reductions translate to local work which in turn keep their respective cores busy, hence strong ordering constraints are not required. Strict-ordered implementations for such cases hinder parallelism.

If the relax-ordered implementation is not selected as optimal then the decision tree selects the strict-ordered implementation by default ($Line\ 17-18$). This selection happens for benchmarks that are not redundant work tolerant. In other words, redundant work is unable to expose parallelism in these benchmarks. They require an implementation that enforces a tight task execution order to eliminate the redundant work. As the goal is to reduce the redundant work, synchronization in these implementations is farily high.

Graph Benchmark (B) and Input (I) Variables

Algorithm 1 employs the high level characteristics of the different implementations and the graph input to optimize the implementation choice. A programmer should be able to transform an implementation and a graph input into these characteristics through benchmarking or visual inspection. To serve this purpose, the Benchmark (B) and Input (I) variables are introduced that quantitatively represent the characteristics of an implementation and a graph input. These B and I variables are borrowed from prior literature [16, 10] and are shown in Figure 1.3.

Different parallelization strategies are used for the graph workloads that dictates the amount of unordered work in the implementation. A graph algorithm usually has an outer loop that goes over the vertices and an inner loop that goes over the edges corresponding to a specific vertex. The outer loop can be parallelized by statically dividing the vertices among all the cores. The benchmark variable vertex division ($B1$) is defined that gives a measure of the percentage of the program in the vertex division. The outer loop can also be parallelized using Pareto execution ($B2$) in which group of vertices distributed among cores statically increase as the benchmark progresses. Alternatively, the inner loop can be parallelized by statically dividing the edges belonging to different vertices among the different cores. The benchmark variable ($B3$) captures this effect by specifying the percentage of the program in the pareto division. A benchmark with large values of any of these variables is highly data parallel and performs high unordered work, thus exposing parallelism. There is no need for a task execution order in such benchmarks as the load can be statically distributed.

The presence of the push-pop operations ($B4$) in a benchmark indicate the presence of the task execution order. The task execution order is there to reduce the amount of redundant work. Push-Pop operations also indicate that the benchmark utilizes queueing primitives and has dynamic load distribution. This means that a task is either enqueued into the local queue or send to a remote core for insertion for the load balancing purposes. Reductions ($B5$) are also sequential operations but they translate to work that is local to a core. In order to protect the data shared among cores, parallel graph algorithms utilize locks or atomics ($B6$). Graph algorithms that don't have shared data are divided into phases. The read-write dependencies need to be forwarded to the next phase so that it can start its execution. These different phases are separated by the barriers ($B7$) to ensure that the dependencies are forwarded and all threads are starting the next phase at the same time. $B6$ and $B7$ collectively are the measure of the synchronization.

All benchmark variables are specified in the percentages therefore they have values ranging from 0 to 1. The values of the benchmark variables $B1$ to $B5$ must add up to 1 as they are collectively specifying the contribution of each phase in the benchmark. For example there might be 60% vertex division ($B1$), 20% push-pop ($B4$) operations and 20% reductions ($B5$) in an implementation. On the other hand, $B6$ and $B7$ are independent of the $B1-5$ variables. $B6$ specifies the percentage of the operations performed on data under atomics or locks in the benchmark and can vary from 0 - 100%. Similarly, $B7$ specifies how frequently barriers appear in the program and can vary between 0 - 100%. A programmer sets the value of these variables by visual inspection or by benchmarking.

The characteristics of an input graph can also be represented in the form of the discrete variables. Figure 1.3 shows the input variables I for the graph input. The size of an input graph is represented by the number of vertices ($I1$) and edges ($I2$).

Higher the number of vertices or edges, easier it is to statically distribute them among the cores, that can process them independently. Another important parameter is the density of the graph which can be calculated using $I1$ and $I2$. Dense graphs usually expose higher parallelism as the edges belonging to a vertex can be statically allocated to the cores. Sparse graphs on the other hand usually have longer dependency chains and prefer implementations with the ordering constraints. In order to identify that a given graph is large or dense, the values of $I1$ and $I2$ should be normalized. This means that the values of $I1$ and $I2$ should be between 0 and 1 so that a threshold of 0.5 can decide whether the provided graph is large or dense. Moreover, vertex and edge counts of some graphs can be extremely large (billion vertices or edges). Therefore, the log normalization is applied on $I1$ and $I2$ instead of the linear normalization. The maximum values of the vertex and edge counts are taken from the prior literature [17].

B-I Variables Based Decision Tree

A decision tree is created in this section, that imitates the inter-implementation selection model but employs the B and I variables to reach the decision of the optimal implementation. Each implementation of the graph benchmark is converted into 7 B variables and input graphs are converted into 2 I variables. The structure of the decision tree is such that first the B variables of the unordered implementation and I variables of the input graph are examined. If the decision tree cannot select the unordered version, then the B variables of the relax-ordered version and I variables of the input graph are analyzed. If the decision tree is still unable to select the relax-ordered variant than the strict-ordered implementation is selected.

Algorithm 2 Inter-Implementation Decision Tree using B variables of different implementations and I variables of input graph

```
Input: float $UO\_B[7]$, float $RO\_B[7]$, float $SO\_B[7]$, float $I[4]$
Output: Optimal Implementation
1: procedure DECISION TREE($UO\_B$, $RO\_B$, $SO\_B$, $I$) return
   $Optimal\ Implementation$
2:     $Optimal\ Implementation$ = ””
3:     $avg\_deg = I[2]/I[1]$
4:     if $((UO\_B[1]\ + UO\_B[2]\ + UO\_B[2])\ > 0.5)$ then
5:         if $(UO\_B[6] < 0.5$ and $UO\_B[7] < 0.5)$ then
6:             if $(I[1] > 0.5$ or $I[2] > 0.5$ or $avg\_deg > 0.5)$ then
7:                 $Optimal\ Implementation$ = Unordered
8:     else if $((UO\_B[1]\ + UO\_B[2]\ + UO\_B[2])\ == 1)$ then
9:         if $(UO\_B[6] < 0.5$ and $UO\_B[7] < 0.5)$ then
10:            $Optimal\ Implementation$ = Unordered
11:    if ($Optimal\ Implementation$ != $Unordered$) then
12:        if $(RO\_B[4]\ > 0.5)$ then
13:            if $(RO\_B[6] < 0.5$ and $RO\_B[7] < 0.5)$ then
14:                if $(RO\_B[5] < 0.5)$ then
15:                    if $(I[1] < 0.5$ and $I[2] < 0.5$ and $avg\_deg < 0.5)$ then
16:                        $Optimal\ Implementation$ = Relax
17:                else
18:                    if $(I[1] > 0.5$ or $I[2] > 0.5$ or $avg\_deg > 0.5)$ then
19:                        $Optimal\ Implementation$ = Relax
20:        if ($Optimal\ Implementation$ != $Relax$) then
21:            $Optimal\ Implementation$ = Strict
```

Algorithm 2 shows the pseudocode of the inter-implementation decision tree based on the B and I variables. The inputs of the decision tree are 7 B variables for unordered ($float\ UO\_B[7]$), 7 for relax-ordered ($float\ RO\_B[7]$), 7 for strict-ordered ($float\ SO\_B[7]$) and 2 I variables for the input graph (*Line* 1) and the output is the optimal implementation. *Lines* $3-7$ in the Algorithm 2 displays the conditions for the selection of unordered versions. The unordered implementation is selected when either of the following condition is met:

- Sum of $UO\_B[1]$, $UO\_B[2]$ and $UO\_B[3]$ of the unordered implementation is greater than 0.5 indicating that the implementation has high vertex or pareto division. High vertex or pareto division shows that the vertices or edges can be statically distributed among the cores. This condition is analogous to checking the presence of high unordered work in the Algorithm 1 required for the selection of the unordered implementations. $UO\_B[6]$ and $UO\_B[7]$ are both less than 0.5 i-e the synchronization is low. In addition to this $I[1]$, $I[2]$ or average degree of the input graph is greater than 0.5 showing that it is a large or dense graph.

- Sum of $UO\_B[1]$, $UO\_B[2]$ and $UO\_B[3]$ is equal to 1. Additionally $UO\_B[6]$ (contention) and $UO\_B[7]$ (barrier) are both less than 0.5 (low synchronization). This means that the unordered implementation regardless of the graph input exposes significantly high parallelism and performs less redundant work, thus it is the optimal choice.

Lines $10 - 18$ in the Algorithm 2 depicts the scenarios for the relax-ordered variant selection. In case when unordered is not selected, relax-ordered is selected when either of the following two conditions is satisfied:

- $RO\_B[4]$ is greater than 0.5 indicating that there are high push-pop operations in the implementation. Push-pop operations, in turn, indicate that the task execution ordered is enforced through queueing primitives to reduce the amount of redundant work. This means that there is limited unordered work. If $RO\_B[4]$ is greater than 0.5, then the sum of $RO\_B[1]$, $RO\_B[2]$ and $RO\_B[3]$ can't be greater than 0.5, which further shows that the amount of the unordered work is low. $UO\_B[6]$ and $UO\_B[7]$ are both less than 0.5 showing that the synchronization is low. Additionally, $RO\_B[5]$ is less than 0.5 i-e there are negligible reductions in the relax-ordered version and the size of the input graph is small i-e $I[1]$ and $I[2]$ are both less than 0.5. Negligible reductions along with the small graph size also suggest that the work local to the core is almost negligible. Hence the relax-ordered version is the optimal choice.

- All the B variables of the relax-ordered implementation (RO_B) have same values as in previous condition except $RO\_B[5]$ which is now greater than 0.5 indicating that the number of reductions in the relax-ordered implementation are high. Additionally, $I[1]$, $I[2]$ or average degree of the input graph is greater than 0.5 depicting that the graph is large or dense. Due to the presence of the significant reductions in the implementation, large or dense graphs perform work that is more native to a specific core. The local work keeps the respective core busy and relaxes the need for strict checks on the task execution order. Thus, the relax-ordered implementation is selected as optimal.

When both unordered and relaxed-ordered are not selected, the strict-ordered implementation is automatically selected ($Lines\ 19-20$). Even though strict-ordered is the default case when decision tree rejects unordered and relax-ordered versions, this happens when:

- $SO\_B[4]$ is approximately equal to 1. $SO\_B[6]$ or $SO\_B[7]$ is greater than 0.5 showing that the synchronization is high. These three parameters show that there are strict constraints on the task execution order which are required for the benchmarks that are extremely less redundant work tolerant.

Optimal Implementation Selection Example on multicore: Figure 1.4 shows the B variables for three different implementations of the SSSP benchmark and the

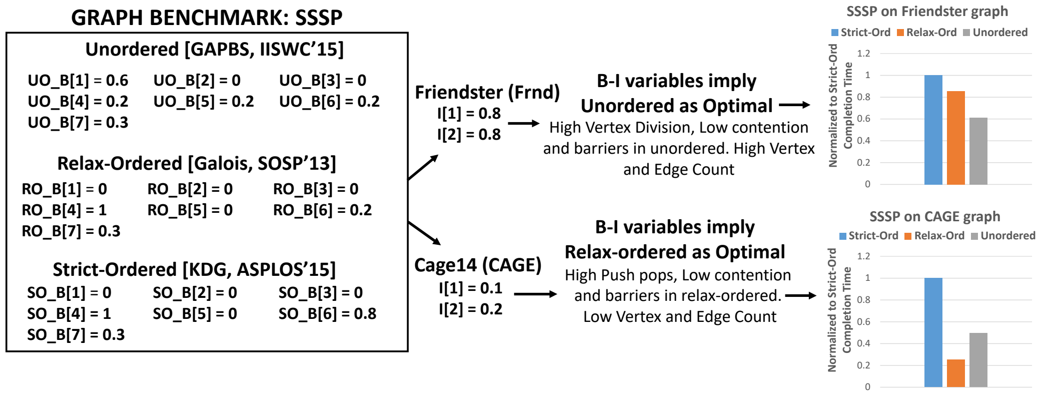
GRAPH BENCHMARK: SSSP
Unordered [GAPBS, IISWC'15]
UO_B[1] = 0.6 UO_B[2] = 0 UO_B[3] = 0
UO_B[4] = 0.2 UO_B[5] = 0.2 UO_B[6] = 0.2
UO_B[7] = 0.3
Relax-Ordered [Galois, SOSP'13]
RO_B[1] = 0 RO_B[2] = 0 RO_B[3] = 0
RO_B[4] = 1 RO_B[5] = 0 RO_B[6] = 0.2
RO_B[7] = 0.3
Strict-Ordered [KDG, ASPLOS'15]
SO_B[1] = 0 SO_B[2] = 0 SO_B[3] = 0
SO_B[4] = 1 SO_B[5] = 0 SO_B[6] = 0.8
SO_B[7] = 0.3
Friendster (Frnd)
I[1] = 0.8
I[2] = 0.8
B-I variables imply
Unordered as Optimal
High Vertex Division, Low contention and barriers in unordered. High Vertex and Edge Count
Cage14 (CAGE)
I[1] = 0.1
I[2] = 0.2
B-I variables imply
Relax-ordered as Optimal
High Push pops, Low contention and barriers in relax-ordered. Low Vertex and Edge Count
SSSP on Friendster graph
Strict-Ord Relax-Ord Unordered
Normalized to Strict-Ord Completion Time
1.2
1
0.8
0.6
0.4
0.2
0
SSSP on CAGE graph
Strict-Ord Relax-Ord Unordered
Normalized to Strict-Ord Completion Time
1.2
1
0.8
0.6
0.4
0.2
0

Figure 1.4: Decision tree workflow for selecting optimal implementation for SSSP with Friendster and CAGE input graphs on Intel Xeon 40-core.

I variables of Friendster [18] and CAGE [19] graphs. $UO\_B[1]$ of the unordered version of SSSP is set to 0.6 as it has high vertex division, $UO\_B[4]$ is set to 0.2 due to the bucketing operations in SSSP delta stepping that translate to the push-pop operations. $UO\_B[5]$ is also set to 0.2 due to presence of some reductions. $UO\_B[6]$ and $UO\_B[7]$ are set to 0.2 and 0.3 as the unordered variant has only atomic operations for the shared $D[]$ array and few barriers for separating different phases. The relax-ordered implementation has $RO\_B[4]$ set equal to 1 while $RO\_B[1]$, $RO\_B[2]$, $RO\_B[3]$ and $RO\_B[5]$ are set equal to 0. This is because of the presence of per-core priority queue for enforcing local order. $RO\_B[6]$ and $RO\_B[7]$ remains the same as for unordered implementation. All B variables of strict-ordered implementation are same as relax-ordered except $SO\_B[6]$ which is high (0.7). This is because strict-ordered implementation enforces global order through an ordered-list shared among the cores that requires frequent locking. For CAGE graph vertex count ($I[1]$) is set to 0.1 and edge count ($I[2]$) to 0.2 as it has very small number of vertices and edges as compare to large graphs such as Twitter. On the other hand $I[1]$ and $I[2]$ for Friendster are set equal to 0.8 as it is a bigger and dense graphs.

Figure 1.4 also shows the workflow of the decision tree that selects the optimal implementation for the SSSP benchmark running two different input graphs (Friendster and CAGE) on the Intel Xeon 40-core machine. The completion times in the figure are normalized to the completion time of the strict-ordered implementation. It can be observed that the unordered version of the SSSP benchmark has high vertex level parallelism ($UO\_B[1] > 0.5$), low Push-Pop operations ($UO\_B[4] < 0.5$) and low synchronization ($UO\_B[6] < 0.5$ and $UO\_B[7] < 0.5$). Due to this **unordered** version performs optimally for Friendster graph as it has high vertex and edge count. The normalized completion times of the different implementations for SSSP-Friendster combination in the figure also reveal that the unordered implementation is the optimal choice. The selection of the unordered implementation for the SSSP-Friendster combination corresponds to *Lines* $3-7$ of the Algorithm 2. On the other hand, CAGE is a smaller graph, so the unordered version is not the right choice for this graph. Observing the B variables of the relax-ordered version of SSSP reveals that it has high Push-Pop operations ($RO\_B[4] > 0.5$), low synchronization ($RO\_B[6] < 0.5$ and $RO\_B[7] < 0.5$) and no reductions ($RO\_B[5] = 0$). Hence, B and I variables imply that the **relax-ordered** implementation is the optimal choice for the SSSP-CAGE combination as choosing unordered version in this case will result in more redundant work. *Lines* $10-15$ of the Algorithm 2 selects the relax-ordered implementation for the SSSP-CAGE combination.

Optimal Implementation Selection Example on GPU: Figure 1.5 shows the workflow of decision tree that selects the optimal implementation for the benchmark input combination of SSSP-Twitter and A*- Twitter on NVidia GTX-1080 GPU. Unlike multicore, GPU has only two implementations (unordered and relax-ordered). The B variables of both of these implementations for SSSP and A* are displayed

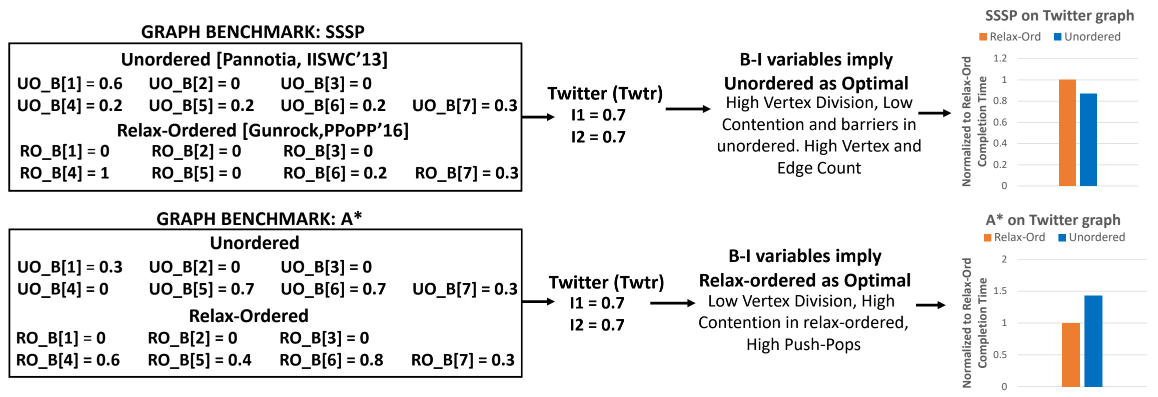
GRAPH BENCHMARK: SSSP
Unordered [Pannotia, IISWC'13]
UO_B[1] = 0.6 UO_B[2] = 0 UO_B[3] = 0
UO_B[4] = 0.2 UO_B[5] = 0.2 UO_B[6] = 0.2 UO_B[7] = 0.3
Relax-Ordered [Gunrock,PPoPP'16]
RO_B[1] = 0 RO_B[2] = 0 RO_B[3] = 0
RO_B[4] = 1 RO_B[5] = 0 RO_B[6] = 0.2 RO_B[7] = 0.3
Twitter (Twtr)
I1 = 0.7
I2 = 0.7
B-I variables imply
Unordered as Optimal
High Vertex Division, Low Contention and barriers in unordered. High Vertex and Edge Count
SSSP on Twitter graph
Relax-Ord
Unordered
Normalized to Relax-Ord Completion Time
1.2
1
0.8
0.6
0.4
0.2
0
GRAPH BENCHMARK: A*
Unordered
UO_B[1] = 0.3 UO_B[2] = 0 UO_B[3] = 0
UO_B[4] = 0 UO_B[5] = 0.7 UO_B[6] = 0.7 UO_B[7] = 0.3
Relax-Ordered
RO_B[1] = 0 RO_B[2] = 0 RO_B[3] = 0
RO_B[4] = 0.6 RO_B[5] = 0.4 RO_B[6] = 0.8 RO_B[7] = 0.3
Twitter (Twtr)
I1 = 0.7
I2 = 0.7
B-I variables imply
Relax-ordered as Optimal
Low Vertex Division, High Contention in relax-ordered, High Push-Pops
A* on Twitter graph
Relax-Ord
Unordered
Normalized to Relax-Ord Completion Time
2
1.5
1
0.5
0

Figure 1.5: Decision tree workflow for selecting optimal implementation for SSSP and A* with Twitter input graph on GTX-1080 GPU.

in the figure along with I variables of Twitter. The unordered version of SSSP has high vertex division ($UO\_B[1] > 0.5$) and low synchronization ($UO\_B[6] < 0.5$ and $UO\_B[7] < 0.5$). Additionally, Twitter has high vertex and edge count ($I[1] > 0.5$ and $I[2] > 0.5$). Due to this **unordered** implementation is selected as optimal for SSSP- Twitter combination. In contrast to this the unordered version of A* has high synchronization ($UO\_B[6] > 0.5$) and low vertex/pareto division ($UO\_B[1] < 0.5$ and $UO\_B[3] < 0.5$). Even though Twitter has high vertex and edge count, unordered can't be selected as optimal due to high synchronization and low vertex/-pareto division. When the unordered is not selected, decision tree simply picks the **relax-ordered** as optimal because there are just two implementations.

The inter-implementation decision tree selects the optimal implementation on a multicore as well as on a GPU. Once the implementation on individual machines are selected, the next step is to compare the performance of the selected multicore implementation with its GPU counter-part. This problem is the choice of an optimal accelerator for a given benchmark-input combination. The HeteroMap framework tends to solve this problem through an offline trained and online deployed deep learning model that optimizes for the accelerator choices (inter and intra). The next section briefly discusses HeteroMap and shows how the inter-implementation decision tree model presented in this section is integrated within the HeteroMap framework.

1.4 HETEROMAP - A PERFORMANCE PREDICTOR

HeteroMap is a state-of-the-art performance predictor that optimizes the inter- and intra- accelerator choices. In the HeteroMap framework, the graph benchmarks and inputs are also represented as the B and I variables which are the superset of the B and I variables introduced in section 1.3. HeteroMap uses percentage of floating point operations, direct accesses, indirect accesses, read-only data, read-write shared data and local data as additional B variables and maximum edge count and diameter as additional I variables. HeteroMap creates a mapping from the B and I variables to the optimal accelerator choices.

Inter- and Intra-accelerator choices

HeteroMap exposes a multi-accelerator setup containing a multicore and a GPU comparable in performance. The framework deploys a deep learning based model that for a given benchmark-input combination chooses either a multicore or a GPU as an optimal accelerator and optimal parameters within these accelerators. A GPU or a multi-core may exhibit superior performance for a given graph benchmark and input due to their diverse capabilities. In GPUs high throughput is delivered by hiding the data latencies through colossal number of threads. The workloads that are highly data parallel with little shared data and small dependency chains can benefit from this capability. A GPU is selected when there is lots of vertex or edge level parallelism in the benchmark as vertices/edges can be divided among the large number of GPU threads. GPU is also selected when the benchmark has high number of reductions, less float point computations and negligible local computations. High synchronization, complex data patterns and inter-thread data movement can severely degrade the performance of GPUs, making them suboptimal for such cases. In order to achieve optimal performance, constraints need to imposed on threading and throughput of a GPU due to the variations in the edge densities of the graph input. This leads to two choices within GPU: *global threading* which is the distribution of threads across GPU chip and *local threading* which is the number of threads on a GPU core.

Multicores give optimal performance when the workloads have complex data patterns due to the presence of cache coherence and cache reuse features. If a benchmark has large number of Push-Pop operations with a high graph density then the multicore is chosen due to the dense graph fitting in its local caches. The multicore is also chosen for benchmarks having reductions and read-write shared data. This is because synchronization required for reductions and shared-data is faster on multicore due to the presence of cache coherence. Benchmarks with indirect addressing modes running larges graphs also perform better on multicore for this reason. Benchmarks having floating point operations with larger graphs also give optimal performance on multicore due to floating point capabilities and stronger cache hierarchy.

Input graph characteristics such as edge densities dictate the intra-multicore parameters such as cores, multithreading, thread placement and SIMD usage. Thread placement is essential for movement of the data along cores and better cache utilization. For example, placing threads closer to the accessed data can improve performance as the data movement and synchronization will be reduced in this case. *KMP blocktime* is another parameter, which specifies the time a thread waits before

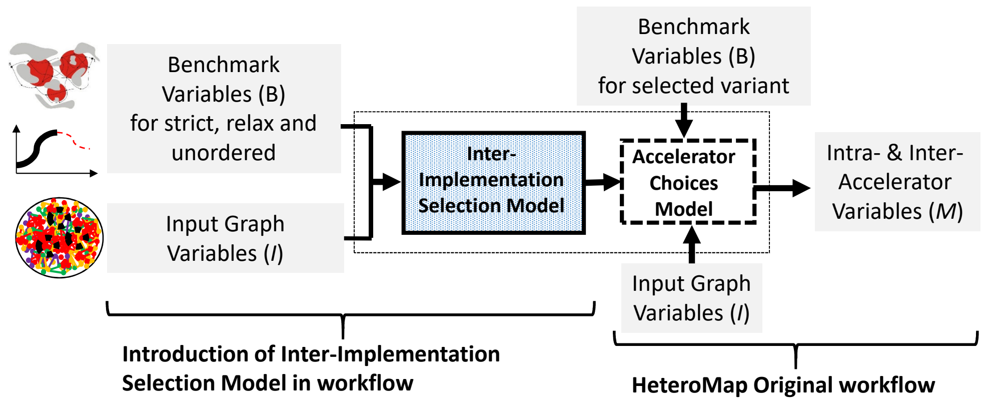
Benchmark Variables (B) for strict, relax and unordered
Input Graph Variables (I)
Inter-Implementation Selection Model
Benchmark Variables (B) for selected variant
Accelerator Choices Model
Input Graph Variables (I)
Intra- & Inter-Accelerator Variables (M)
Introduction of Inter-Implementation Selection Model in workflow
HeteroMap Original workflow

Figure 1.6: Modified HeteroMap Setup to incorporate Inter-Implementation Selection Model

going to sleep. Increasing this time in case of high contention and load imbalance, can improve performance. Apart from these parameters, there are OpenMP specific parameters exposed by HeteroMap on multicore. They consist of thread scheduling with different data tile/chunk sizes, nested parallelism and thread wait time for OpenMP calls.

Inter-Implementation Model Intergration with HeteroMap

HeteroMap chooses the optimal accelerator for the given benchmark-input combination but fails to differentiate between different implementations of a benchmark. This is because HeteroMap was originally trained for unordered versions of the benchmarks only and unordered implementations don't exhibit best performance for all benchmark-input combinations. Due to this HeteroMap can benefit from the inter-implementation model because it makes HeteroMap capable of selecting the optimal implementation with optimal accelerator for the given benchmark-input combination.

Figure 1.6 shows the modified setup of HeteroMap that includes the inter-implementation selection model. In the original workflow of HeteroMap benchmark

(B) and graph input (I) variables are passed to a deep learning model that selects the optimal accelerator and different parameters related to that accelerator. In the modified HeteroMap workflow, the inter-implementation decision tree module is introduced between inputs (benchmark and graphs) and inter- and intra-accelerator model. As there are three different implementations for each benchmark, B variables of each implementation and I variables of a graph are provided as an input to the inter-implementation selection model. The inter-implementation decision tree selects the optimal implementation on a multicore and a GPU and passes this decision to inter- and intra-accelerator model . The inter- and intra-machine model takes B and I variables of the selected implementation and selects the optimal accelerator as well as the optimal intra-accelerator choices for a given benchmark-input combination. In this way HeteroMap can not only select the optimal accelerator but also the optimal implementation.

1.5 METHODOLOGY

Machine Configurations

In order to evaluate the inter-implementation decision tree, first a single-accelerator setup (a multicore or a GPU) is considered. The decision tree is evaluated on both machines separately. After that the inter-implementation decision tree is integrated within HeteroMap and a multi-accelerator setup containing a multicore and a GPU is used. The multicore machine is the Intel Xeon E5-2650 v3 machine having 10 hyperthreaded cores running at 2.30 GHz, with a 1TB DDR4 RAM. GPU used in this work is NVidia GTX-1080. It comprises 2560 cores with 8.8 TFLOPs single-precision and 0.27 TFLOPs double-precision compute capability, and has 8GB memory size.

Table 1.1: 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 1.2: 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*)	

Parameters of the two accelerators are shown in Table 2.1

Benchmarks and Inputs

The graph benchmarks evaluated in this work are divided into two categories. The benchmarks in the first category have three different implementations namely strict-ordered, relax-ordered and unordered. The benchmarks in the second category have only unordered implementations. The benchmarks belonging to the two categories are listed in Table 2.2. Inter-Implementation selection model is applicable for the first category of the benchmarks. However, inter-accelerator selection model is applicable for both categories of benchmarks.

Multicore Benchmark Implementations: The strict-ordered and relax-ordered variants of the benchmarks in the first category are acquired from KDG [9] and Galois [10] respectively with the exception of A* as there is no implementation of A* in KDG or Galois. The strict-ordered and relaxed-ordered version of A* are implemented within the KDG and Galois frameworks. The heuristic for A* is set equal to 0 as there is no additional information available with the input graphs and the destination node is chosen randomly. The unordered variants of SSSP and MST are taken from the GAP benchmark suite [20] and the problem based benchmark suite (PBBS) [21] respectively. The unordered versions of BFS, Connected Components, Coloring, DFS, PageRank, PageRank-DP, Triangle Counting and Community are acquired from CRONO [22]. Unordered version of A* is implemented within the CRONO suite.

GPU Benchmark Implementations: The relax-ordered implementations of the benchmarks in the first category are taken from the Gunrock framework [23]. There are no GPU versions of strict-ordered implementations. The unordered version of SSSP, BFS, Graph Coloring, PageRank and PageRank-DP are acquired from Pannotia [7] and Rodinia [24]. The unordered variants of the remaining graph benchmarks are ported from multicore to GPU within the Pannotia suite using OpenCL.

Graph Inputs: Diverse input graphs are utilized for the evaluation of this work, varying from sparse to dense and from smaller in size to larger. They are obtained from the different domains such as road networks, social networks and biological networks. Table 2.3 displays the input graphs with their vertices count, edges count and respective sizes in gigabytes. The input graphs that are unable to fit in accelerator's main memory, are divided into smaller spatio-temporal chunks using the Stinger Framework [25]. These chunks are processed sequentially. Memory transfer times are not included in the completion times for the fair comparison between different accelerators. Final completion time only contains the accelerator execution time and overhead of the proposed predictor.

Table 1.3: Graph Inputs used for Evaluation

Graph	$\|V\|$	$\|E\|$	Size (GB)
USA-CAL (CAL) [26]	1,890,815	4,657,742	0.1
Cage14 (CAGE) [19]	1,505,785	27,130,349	0.1
Facebook (FB) [27]	2,937,612	41,919,708	0.3
Livejournal (LJ) [28]	4,847,571	92,115,620	0.7
com-Orkut (Orkt) [17]	3,072,627	234,370,166	4
Twitter (Twtr) [5]	41,652,231	1,468,365,182	27
Friendster (Frnd) [18]	124,836,180	3,612,134,270	52

Performance Metrics

The output of the proposed predictor is compared with the ideal output that represents the ground truth i-e the true optimal implementation for all benchmark-input combinations. The ideal predictor is the one that achieves the maximum possible performance gains by selecting the right implementation. The percentage differences in the performance acquired through proposed predictor and the ideal predictor are also quantified on single accelerator setup for inter-implementation selection model. The accuracy of the inter-implementation decision tree is measured by the percentage difference in performance gains. For Top-1 accuracy, if the percentage difference in performance for a given benchmark-input combination is within 1%, the choice is considered to be optimal. For Top-10 accuracy, percentage difference in performance should be less than 10%.

Once the inter-implementation decision tree is individually evaluated on a multicore and a GPU, it is then evaluated in the context of HeteroMap's multi-accelerator environment. HeteroMap with inter-implementation decision tree (Modified HeteroMap) selects the optimal implementation and accelerator for all benchmark-input combinations. Modified HeteroMap performance is compared with the multicore only

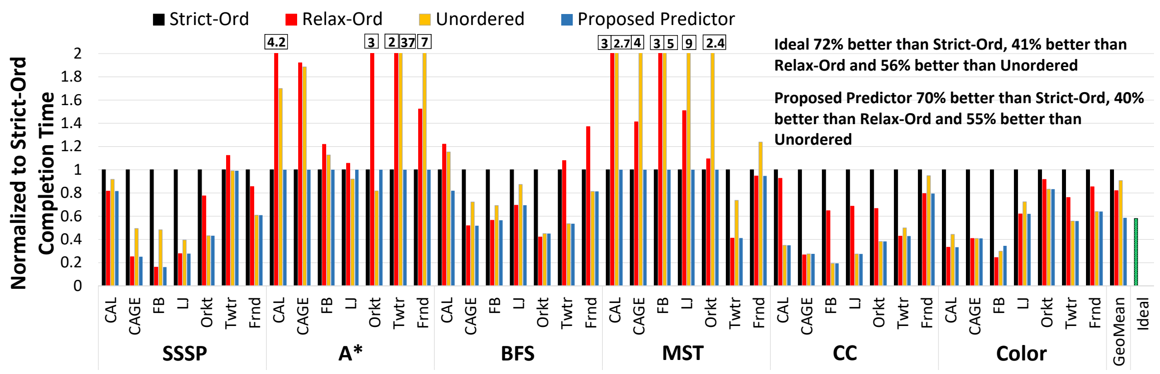

Figure 1.7: Proposed predictor performance comparison for graph benchmark-input combinations on multicore. All results are normalized to Strict-ordered implementation completion time (Higher is worse)

and the GPU only baselines. Modified HeteroMap performance is also compared with the performance of Original HeteroMap. The percentage difference in performance is used as a metric in both of these comparisons.

1.6 EVALUATION

This section evaluates the proposed inter-implementation predictor on the Intel Xeon E5-2650 v3 multicore and a NVidia GTX-1080 GPU individually. The predictor is also evaluated in the context of HeteroMap.

Inter-Implementation combinations on multicore

Figure 1.7 shows the performance variations for all the benchmark-input combinations on Intel Xeon E5-2650 v3. All completion times are normalized to their respective strict-ordered implementation's completion times. The benchmarks with high vertex or edge level division such as SSSP, BFS and Color along with the large and dense input graphs such as Twitter (Twtr), Friendster (Frnd) and Orkut (Orkt) prefer unordered implementations. This is because these benchmark-input combinations have enough independent tasks that can be executed in parallel. Enforcing ordering constraints extract no performance gains in these combinations as the amount of the unordered work is significantly high. The relax-ordered implementations are selected when the same benchmarks (SSSP, BFS and Color) are run with smaller or sparse graphs such as USA-CAL (CAL), Cage14 (CAGE), Facebook (FB) and Live-Journal (LJ). As the graphs are smaller, a task execution order is required to reduce the amount of redundant work. The relax-ordered implementations are also preferred for the benchmarks with less vertex/edge level parallelism and some reductions such as MST running large and dense input graphs. The strict-ordered implementations are regarded optimal for the benchmarks that are little redundant work tolerant and require high synchronization such as A*. The output of the A* search is a single path from the source to the destination, hence the parallelism available is limited. The strict-ordered implementations perform better in these cases as they ensure that their work efficiency is same as that of the sequential version. MST with smaller graphs also prefer the strict-ordered implementations for the same reason. The proposed predictor is able to select the ideal implementation for 95% (Top-1 accuracy) of the benchmark-input combinations. In cases where proposed predictor is unable to choose the optimal implementation e.g BFS-Orkt, MST-Orkt, performance difference between the ideal and the selected is within 10%. Therefore the Top-10 accuracy of the predictor is 100%. If only the strict-ordered, relax-ordered or unordered is selected for all benchmark-input combinations then the ideal predictor (one that always chooses the best performing version) is 72% better than the strict-ordered, 41% than the relax-ordered and 56% better than the unordered implementations. The proposed predictor is 70% better than the strict-ordered, 40% than the relax-ordered and 55%

better than the unordered variants.

Inter-Implementation combinations on GPU

Figure 1.8 shows the performance variations for all benchmark-input combinations on NVidia GTX-1080 GPU . All completion times are normalized to their respective relax-ordered implementation's completion times. The benchmarks with high vertex/edge division (SSSP, BFS and Color) executing larger and dense graphs (Twitter, Orkut and Friendster) prefer unordered implementations as in case of multicore. Contrarily, smaller and sparse graphs running with the same benchmarks again lean towards the relax-ordered implementations like multicore. As there are no strict-ordered implementations for the benchmarks on GPU, all benchmark-input combinations that map to the strict-ordered on multicore, choose relax-ordered as the optimal implementation. Examples of such cases are A* benchmark with any input graph and MST with smaller and sparse graphs (CAL, FB etc). The relax-ordered implementation of A* with all input graphs shows huge performance gain (2x) over unordered implementation. This is because A* has to find a single path from the source to the destination and to begin with, it has limited parallelism. The unordered implementation of A* fails to expose parallelism at the expense of high redundant work. Similar trends can be seen for MST as well. Alternatively, the unordered implementation of Connected Component particularly with CAL, CAGE and FB displays better performance (63.2% for CAL, 36.1% for CAGE and 38.4% for FB) over its relax-ordered counter-part with the same input graphs. The reason for this is that there is plenty of parallelism available in Connected Componented that is better exploited by the unordered implementation. The Top-1 accuracy of the proposed predictor on

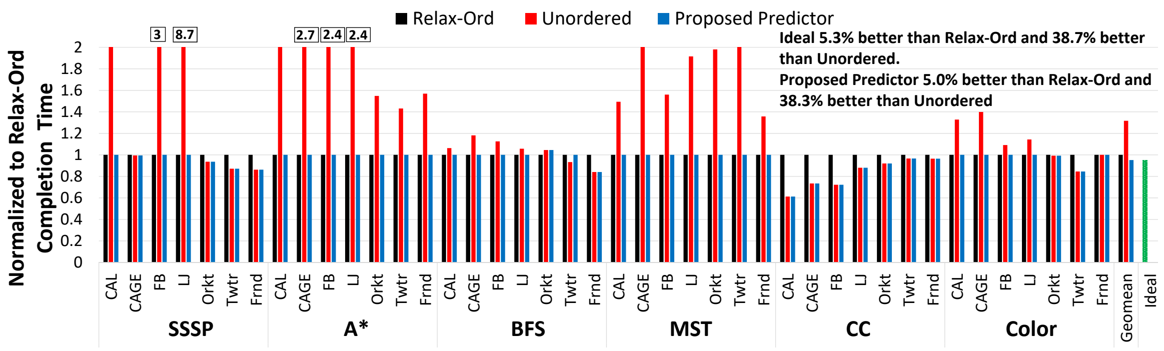

Figure 1.8: Proposed predictor performance comparison for graph benchmark-input combinations on GPU. All results are normalized to Relax-ordered implementation completion time (Higher is worse)

GPU is 97.6% as it captures the optimal implementation for all benchmark-input combinations except one (BFS-Orkt). However, the performance difference between the selected and the optimal implementation for this combination is less than 10%, hence the Top-10 accuracy of the proposed predictor on GPU is 100%. The ideal predictor is 5.3% better than choosing only the relax-ordered implementations for all benchmark-input combinations and 38.7% better than choosing only the unordered implementations while the proposed predictor is 5.0% better than relax-ordered and 38.3% better than unordered.

Modified HeteroMap Performance Evaluation

Figure 1.9 shows the performance comparison of Intel Xeon multicore and GTX-1080 GPU for all benchmark-input combinations. First six benchmarks starting from the left in the figure (SSSP to Color) have three different implementations (strict, relax, unordered) on multicore and two (relax, unordered) on GPU for each benchmark. Only the completion times for the optimal implementation on multicore and GPU are shown as the inter-implementation decision tree has already chosen the best performing version. Last five benchmarks (DFS to Community) in the figure only have unordered versions, hence that is the optimal implementation for them on multicore and GPU.

GPU Combinations: Highly data parallel workloads such as BFS, DFS and Connected Component give superior performance on GPU. This is because they have high vertex/pareto division, extremely low read-write shared data, low synchronization, and simple data access patterns. These benchmarks have large number of independent tasks that are statically distributed among the GPU threads. Due to the negligible inter- and intra- task dependencies, tasks do not require execution order. The massive number of GPU threads can extract high parallelism for these benchmark-input combinations. One notable exception is BFS-CAGE combination that performs significantly well on multicore as compare to GPU. This is because CAGE graph has high disparity in the number of edges belonging to a specific vertex. The GPU implementation suffers as the number of edges at a certain level may not be large enough for the threads to extract parallelism.

Multicore Combinations: The benchmarks that benefit from the task execution order to reduce the amount of redundant work fare well on multicore. This is because ordering constraints are enforced through queueing constructs and push-pop operations work better on multicore due to strong cache hierarchy. This is essentially true for benchmarks for which the inter-implementation decision tree selects the relax-ordered or strict-ordered versions such as MST, A* and some combinations of SSSP (with smaller or sparse graphs). They both enforced local order through per core priority queues. There are some combinations of SSSP with larger graphs (Twitter, Orkut and Friendster) for which the inter-implementation decision tree selects unordered as optimal. Even for these benchmark-input combinations, multicore is selected. This is because Delta-Stepping is used as the unordered version instead of Bellman Ford for SSSP. Delta-Stepping has high read-write shared data and synchronization as compare to Bellman Ford but performs less redundant work. It performs better on multicore as compare to GPU due to faster inter-thread communication and better synchronization operations.

The graph algorithms that contain high floating point operations also give superior performance on multicore. The benchmarks such as PR, PR-DP and Community run better on multicore as Intel Xeon is better than GTX-1080 GPU in terms of double precision floating-point operations. The unordered benchmarks with high contention such as Triangle Counting also perform better on multicore. This is because synchronization primitives on multicore benefit from cache coherence. One notable exception is the TC-CAGE combination, it performs better on GPU as it has less synchronization due to extreme sparsity in some region of the CAGE graph. The Top-1 accuracy of the inter-accelerator decision tree of modified HeteroMap is 95.45% and the Top-10 (performance difference with in 10%) accuracy is 98.48%. Modified HeteroMap geometrically performs 15% better than the multicore only and 42% better than the GPU only as compare to ideal which performs 17% better than multicore and 45% better than GPU.

Comparison with Original Heteromap Figure 1.10 shows the performance comparison of modified HeteroMap (with inter-implementation selection model) with original Heteromap for all benchmark-input combinations. All completion times in the figure are normalized to original HeteroMap's completion times. Last five benchmarks in the figure (DFS to Community) show no performance gain over original Heteromap as they have only unordered implementations. They are unable to benefit from the inter-implementation selection model. All of the performance gains are in

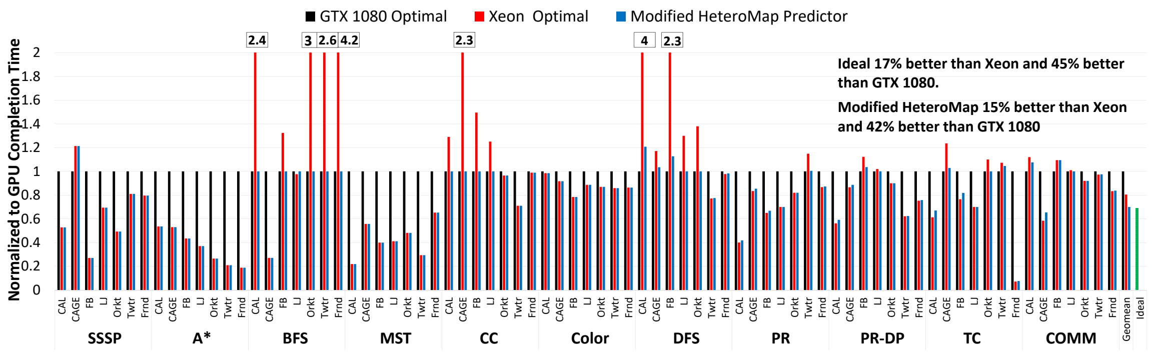

Figure 1.9: Modified HeteroMap performance comparison for graph benchmark-input combinations on multiaccelertor setup. All results are normalized to GPU completion time (Higher is worse)

the first six benchmarks (SSSP to Color). The benchmark-input combinations that select the relax-ordered or strict-ordered implementations as optimal on multicore exhibits performance gains over original HeteroMap such as SSSP and Color with smaller graphs (CAL, FB and LJ), and MST and A* with all input graphs. On GPU, benchmark-input combinations that select the relax-ordered versions as optimal show superior performance as compare to original HeteroMap such as BFS with almost all graph inputs. In the first six benchmarks (SSSP - Color), only CC fails to deliver better performance than original HeteroMap. This is because the unordered implementation of CC on GPU is chosen as the optimal implementation for all graph inputs. As original HeteroMap already optimizes for the unordered implementations, no performance gains are seen. On average modified HeteroMap displays performance gain of 31.5% over original HeteroMap. This is due to the integration of implementation choice in HeteroMap.

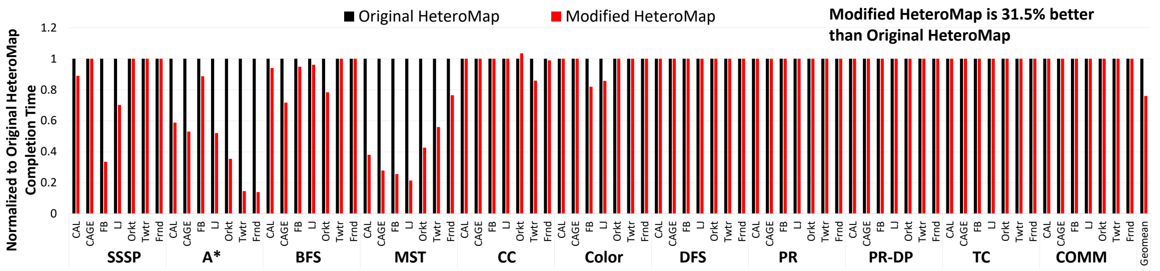

Figure 1.10: Performance Comparison of Modified HeteroMap with Original HeteroMap for graph benchmark-input combinations. All results are normalized to Original HeteroMap completion time (Higher is worse)

2 EXPLORING PARALLEL IMPLEMENTATION CHOICES FOR TEMPORAL GRAPHS

2.1 MOTIVATION

Graph algorithms are widely employed in today's real world due to their enormous number of applications in different domains. They are key components of various systems from diverse domains such as traffic map applications [4], self-driving cars [3], social network analytics, and routing algorithms [5]. From the perspective of the 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, the graph algorithms are designed for the input graphs that are static in nature. The properties of the static graphs don't evolve with time i.e. the number of vertices, edges and the weights of the edges are not the 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 [29], social networks [30], biological sciences [31] and wireless sensor networks [32] 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 [33]. Due to the omnipresent applications of time-varying graphs in various domains, it is important to study the behavior of different graph algorithms processing time-varying graphs.

There is plenty of research present on time-varying graphs in the prior literature covering various aspects such as the representation and modeling [34, 33, 35], and the analysis of the temporal graphs [36, 37, 38, 39]. On the algorithmic front, traditional graph workloads such as the single shortest path problem [33, 40, 41, 42, 43], breadth and depth first search [44], minimum spanning tree [45], page-rank [46, 47] and community detection [48] are proposed for temporal graphs. Even though, there is a plethora of research on the temporal graph algorithms, performance analysis of the parallel algorithms in terms of optimal implementation (parallel implementation that gives best performance) on different accelerators such as a multicore and a GPU tends to be overlooked in the 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 counter-parts. The absence of the performance implication studies from the prior literature is also due to the limited availability of the temporal graph datasets. Additionally, the available temporal graphs are smaller in size [17] as compared to their static counter-parts hence limiting the opportunities for exploiting parallelism.

In order to study the performance of the temporal graph algorithms on different accelerators, there is a dire need for the 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) [26] or biological graph (Cage14) [17] can be converted into temporal graphs. This motivates the need for a tool that can take a static graph as input and outputs its temporal version by systematically adding edges at specific time instants to represent a real temporal graph. The tool should also be capable of varying the weights of the edges at different time instants as the output of some graph workloads such as SSSP and A* is sensitive to the value of weights.

The conversions of the available static graphs into temporal graphs provide an opportunity to explore the performance aspects of the various temporal graph benchmarks on different accelerators. The sequential version of graph workloads for static graphs such as SSSP are implemented using queueing primitives to enforce task execution order. The parallel implementations of the graph benchmarks can be decided into three different categories based on the ordering constraints. The unordered parallel implementations of these benchmarks 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 counter-part, strict-ordered implementations additionally maintain global order at the expense of high synchronization. Benchmarks running temporal graphs can also have these three different parallel implementations.

The optimal implementation for a temporal graph benchmark and input is the one that gives the best performance on a certain accelerator. The variations in the graph benchmark leads to different optimal implementations on an accelerator for example temporal SSSP and A* running temporal version of California road network graph may prefer different parallel implementations. Additionally, the changes in temporal graphs also effect the performance of the parallel implementation and hence the choice of the optimal implementation. Similarly, the choice of the optimal accelerator is also dependent on the graph benchmarks and inputs. A certain temporal benchmark-input combination may give superior performance on a multicore and another benchmark-input combination may prefer GPU. This motivates the need for the exploration of the optimal implementation and accelerator choice space for the given temporal benchmark-input combinations. The contributions of this work are outlined below:

- A conversion tool is proposed and implemented that transforms static graphs into temporal graphs. This allows to explore the performance aspects of the graph benchmarks running temporal graphs.

- The optimal implementation choices on different accelerators for temporal graph benchmarks and inputs are explored. It is shown that selecting the optimal implementation for all benchmark-input combinations gives a geometric performance gain of 46.38% on Intel Xeon 40-core and 20.30% on NVidia GTX-1080 GPU.

- The optimal accelerator choices for temporal graph benchmarks and inputs are also studied. It is shown that selecting the optimal accelerator for all benchmark-input combinations exhibits a geometric performance gain of 30%.

- The performance of temporal graphs are also compared with their static counterparts and it is shown that the choice of the optimal implementation and accelerator for a temporal and its corresponding static graph is not always the same.

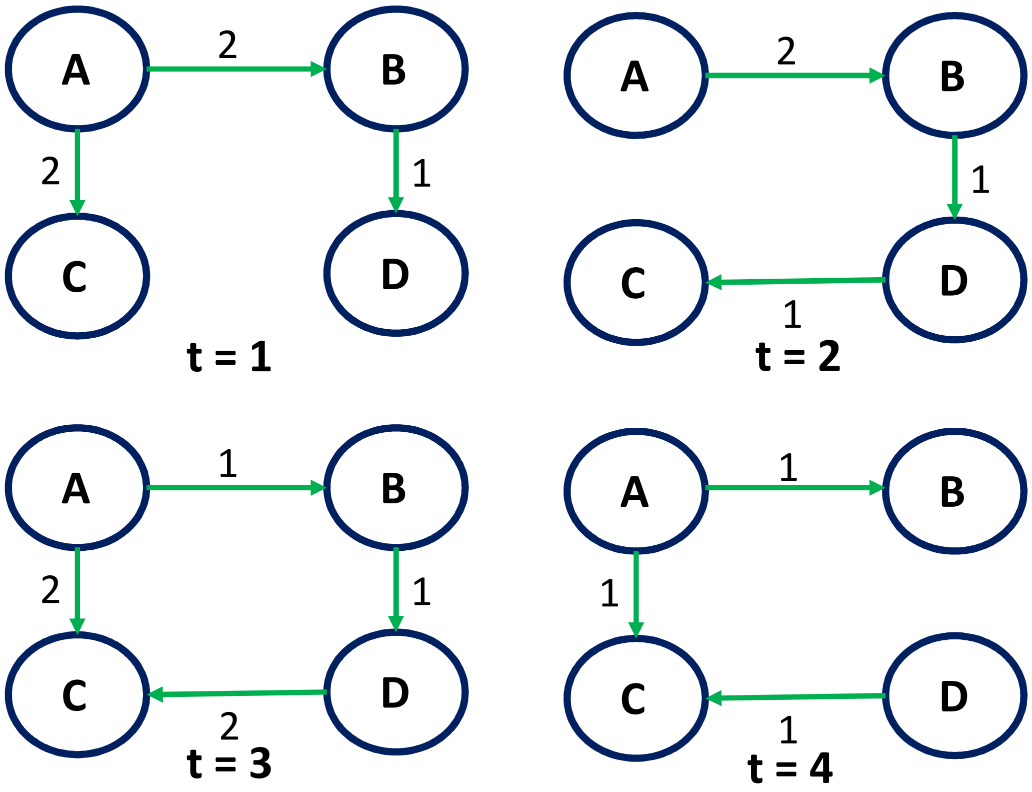

Figure 2.1: Snapshots of a directed and weighted temporal graph at four time instants

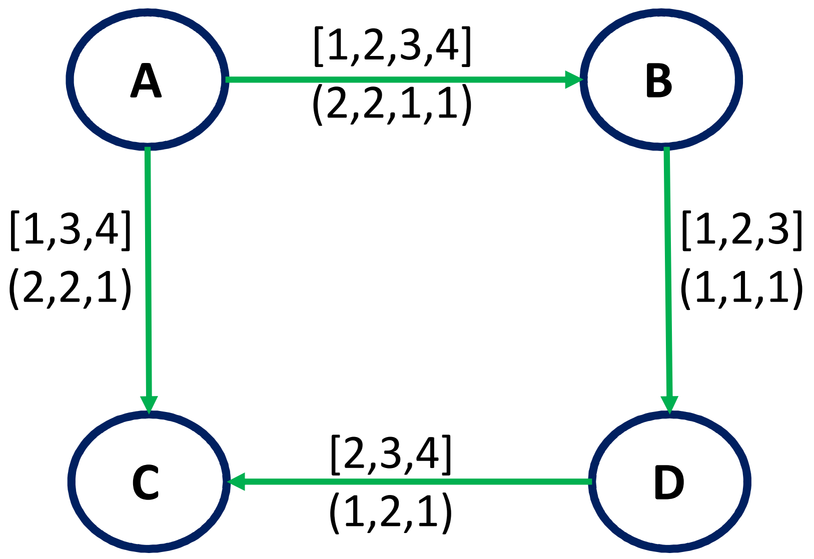

Figure 2.2: Time aggregated graph representation of a temporal graph

2.2 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 the edges are also a function of time and have different values at different time instants. Figure 2.1 shows an example of a directed and weighted time-varying graph at four different time instants. This graph contains four vertices and are connected through time-dependent edges. At different time instants, the total number of edges in this temporal graph vary as the edges are appearing and disappearing with 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.

In order to represent different snapshots of a temporal graph in one concise representation, B George et.al proposed time-aggregated graphs [33]. Time aggregated graphs represent the temporal graphs by capturing their characteristics through time series. There are two series associated with each edge of the 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 2.2 displays the time-aggregated graph representation of the temporal graph shown in Figure 2.1. 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 the temporal graphs is to expand the time along 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 aggregated graphs are equal to 1. Figure 2.3 displays the time-expanded graph representation of the temporal graph shown in Figure 2.1. 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 the 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 benchmark performs more work on the time-expanded graph as compare to its time-aggregated counter-part. As the redundant work can immensely affect the performance of the parallel implementation of the graph benchmark, time-aggregated graphs tend to be

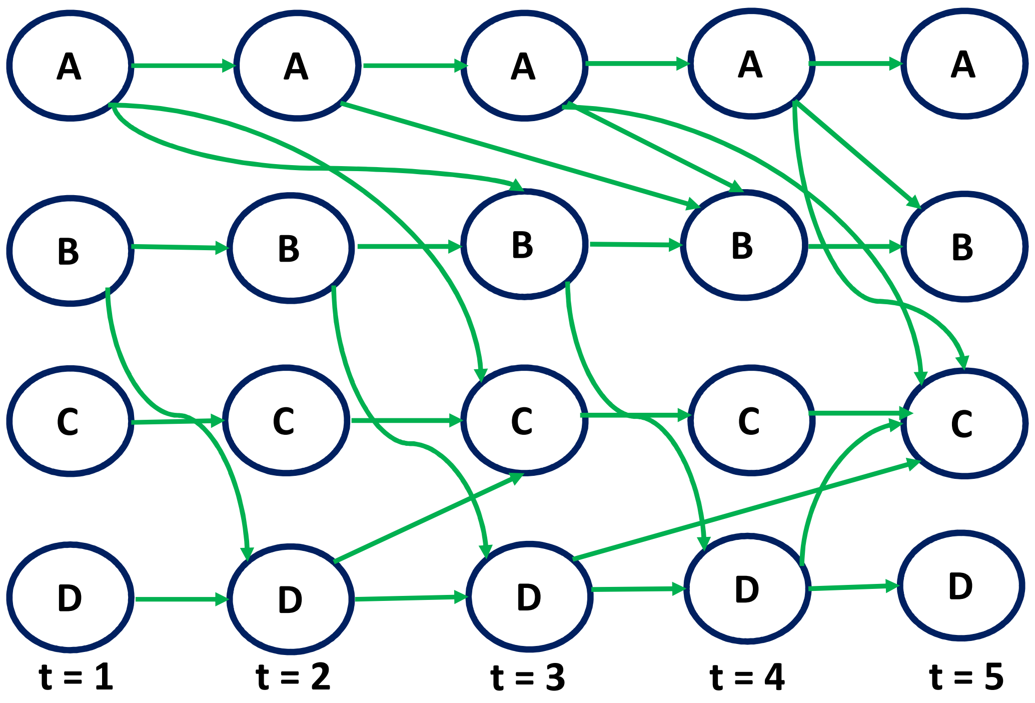

Figure 2.3: Time expanded graph representation of a temporal graph

a better and compact way of representing the temporal graphs. In the evaluation section, a study is presented that shows the performance advantage of time-aggregated graphs over time-expanded graphs for different graph benchmarks.

2.3 TEMPORAL GRAPH GENERATION

This section presents a tool that is capable of converting a static graph into a temporal graph. Any static graph can be viewed as a temporal graph having all the 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 the edges for various time instants.

One simple and naive method to add the temporal edges to a static graph is to randomly decide whether an edge should be added for a specific time instant or not. Algorithm 3 shows the pseudocode for the temporal graph generation utilizing this simple strategy. The algorithm takes as input the static graph and the maximum number of time instant for the edge and outputs a temporal graph. It has an outer loop that goes over all the vertices of the static graph and then 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 will be added for that time instant otherwise not. The weight of all the 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 3 Naive algorithm for temporal graph generation

```
Inputs: $G(V, E) \leftarrow$ Static Graph, $T \leftarrow$ Maximum time instants for an edge
Outputs: $G^t(V, E^t, t) \leftarrow$ Temporal Graph
1: for (each vertex $v \in V$) 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 3 is able to generate a temporal graph by using a simple method, however the generated temporal graph doesn't imitate a real temporal graph. The problem with the algorithm is that it is treating edges belonging to different vertices equal as it always samples from the uniform distribution between 0 and 1. Additionally, all the time instants are also treated equally and the weight of a temporal edge is 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, lets take example of USA California road-network graph. The traffic patterns for the big cities such as San Francisco are different from the traffic pattern of a suburb e.g. Pleasanton. Similarly, the traffic patterns during the rush hours are also different than the patterns during the normal hours. Due to this the weights of the edges in a temporal graph should increase during rush hour and should decrease during the normal hours. There is a need to change Algorithm 3 in a way that for different parts of the graph and for different time instants, sampling is done from a different probability distribution.

Algorithm 4 Improved algorithm for temporal graph generation

```
Inputs: $G(V, E) \leftarrow$ Static Graph, $T \leftarrow$ Maximum time instants for an edge
$n \leftarrow$ number of vertex's sets, $m \leftarrow$ number of time intervals
Outputs: $G^t(V, E^t, t) \leftarrow$ Temporal Graph
1: for (each vertex $v \in V$) 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($v, u, w\_t, t$)
```

Algorithm 4 shows the pseudocode for an improved version of algorithm 3 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 Node ID but if any additional information is available with the graph that can be used to achieve this partition. For example, USA-CAL 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 (maximum number of time instants). This is done to achieve the temporal paritioning of the graph. The loops for vertices, edges and time instants remain the same as in Algorithm 3. For every vertex the algorithm determines its corresponding set and for every time instant the algorithm determines the time interval it belong to. The value of set and time interval are passed to the *sample_from_probability_distribution* method so that it can sample from related probability distribution. As there are n set of vertices and m set of time instants, there are total of $m * n$ probability distribution. 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.

Utilizing the tool mentioned in this section, the static graphs available in literature are converted into temporal graphs. Various graphs from different sources ranging from smaller in size to larger and sparse to dense are used for the conversion into temporal graphs. The generated temporal graphs and their different characteristics are shown in the methodology section.

2.4 ALGORITHMIC-CENTRIC CLASSIFICATION OF PARALLEL TEMPORAL GRAPH BENCHMARKS

Exploiting task-level parallelism makes parallel programming simple and has gained popularity due to its integration in the modern parallel programming frameworks [9, 10]. In order to expose parallelism, the parallel implementation of workload specifies a unit of execution called task. that performs some fixed operations and executes in parallel with other similar 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 accelerator. A task execution order is present in all task parallel algorithms. The order is required as the execution of a particular tasks depend 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 the thread synchronization becomes an important component of the execution model as inter- and intra- task dependencies benefit from it. 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.

Algorithm 5 Generic pseudocode of the strict-ordered implementation

```
$tid \leftarrow$ Core ID
$PQ[tid] \leftarrow$ Priority Queue for each core
$TaskList \leftarrow$ 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 the task execution. Due to this they have the work efficiency of their sequential counter-parts.

Extracting parallelism while at the same time following a strict global order is an extremely difficult problem. The Kinetic Dependence Graph(KDG) [9] framework is one such example that exploits task-level parallelism while at the same time enforces 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 the task insertion and deletion. When a task is dequeued from the priority queue of a core, each core runs the safe-source-tes,t that finds the dependency of the dequeued task on the tasks in other core through ordered list. The dequeue operation stalls if there is a dependecy detection untill that dependency is reolved. However, the tasks execute in parallel if no dependencies are found. The goal is to find tasks that can be executed in parallel with each on different cores to exploit parallelism. Algorithm 5 shows the pseudocode for a generic strict-ordered implementation of the graph benchmark. A local queue is present on each core and there is a global ordered list shared among all the core. Each 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 indicate 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 child of the tasks and then for all the time instants of the child to create new tasks. These tasks are either inserted into the local priority queue or send to a remote core for load balancing purposes. After the execution of critical section, each task is atomically added to the shared ordered list.

Due to to the presence of global order at the cost of high syhncronization cost in strict-ordered algorithms parallelism exposed is extremely limited. Alternatively large amount of parallelism can be exposed by performing redundant work. This

Algorithm 6 Generic pseudocode of the relax-ordered implementation

```
$tid \leftarrow$ Core ID
$PQ[tid] \leftarrow$ 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)
```

approach is adopted by the Galois [10] framework by removing the global ordering constraints enforced by strict-ordered implementations of KDG and having local priority queues per core for local order. As there is no order list shared among the cores, the synchronization cost is extermely reduced as compare to strict-ordered implementations. However, the amount of redundant work is high in relax-ordered implementations as compare to strict-ordered versions as they have to pass through multiple iterations before coming to convergence. Algorithm 6 shows the pseudocode for a generic relaxed-ordered implementation of the graph graph benchmark. There is 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, 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 childern of the task and inner loop goes through time instants of each child and new tasks are created. The created tasks are either inserted into the local priority queue or send to a remote core.

Unordered task parallel algorithms remove both the local and global task execu-

Algorithm 7 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)
```

tion order. They allow different tasks to execute in parallel with other task and are implemented in such a way that inter-task dependencies are minimized. Due to the absence of the local and global order, unordered implementations pass through large number of iterations before they converge. The large number of iterations in turn may 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 compare to their relax-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 the data dependencies are forwarded properly and the next phase of the implementation can be safely initiated by each thread. Algorithm 7 displays the pseudocode for a generic unordered implementation of a task parallel workload. Each core has an unordered list (instead of priority queues) that contains the tasks to be executed. Tasks present in the task list of different cores can be executed in parallel with each other. Each core gets a task from its task list then runs a loop over all the childern of the task and the time instants of the childern. 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 trade-offs present between the parallelization strategy, synchronization and work efficiency in the three different implementations of a graph benchmark, as revealed by their pseudocodes:

- Vertices or edges are statically distributed among the threads when the local and global 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 at the same time exposes significant parallelism.

- Optimal work efficiency can be achieved by maintaining a global order at the expense of high synchronization cost. However, this immensely reduces the amount of 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 input can be the primary sources 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 they are tied back to these high level-characteristics of the temporal graph benchmarks.

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

Table 2.1: 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

counter-parts. Due to the presence of this time instants loop, the amount of exposed parallelism increases. As there is higher parallelism in temporal implementations, the choice of optimal implementation for temporal benchmarks and graphs may not be same as their static counter-parts. In the evaluation section, the choices for static graph benchmarks and inputs are evaluated for temporal combinations.

2.5 METHODOLOGY

Machine Configurations

Inter-implementation choices for different static and temporal benchmark input combinations are evaluated individually on a multicore and a GPU. Inter-accelerator choices employs a multi-accelerator setup (a multicore and a GPU) for the evaluation. Intel Xeon E5-2650 v3 is used as a multicore machine. It has 10 hyperthreaded cores clocking at 2.30 GHz and a 1TB DDR4 RAM. This work utilizes NVidia GTX-1080 GPU for the evaluation. The GPU has 2560 cores with 8.8 TFLOPs single-precision and 0.27 TFLOPs double-precision compute capability, and has 8GB memory size. Table 2.1 show different parameters of the two machines.

Table 2.2: 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*)	

Benchmarks and Inputs

Graph workloads 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 first category. In the second category the benchmarks only have unordered implementations. Table 2.2 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 seond catgeory only has one implementation. However, the inter-accelerator choices are valid for both categories of the benchmarks as all of the graph workloads can have atleast 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 [20] and the problem based benchmark suite (PBBS) [21] respectively. The unordered variants of Connected Components, BFS, DFS, PageRank, PageRank-DP, Coloring, Triangle Counting and Community are taken from CRONO [22]. The unordered variant for the A* workload is implemented in the CRONO suite. The strict-ordered and relax-ordered variants of the benchmarks in the first category are acquired. The KDG [9] and Galois [10] frameworks provide the relax-ordered and strict-ordered implementations of the benchmarks falling in first category, except A* which 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 two implementations of a benchmark for GPU. GPU doesn't has strict-ordered implementations. The Gunrock framework [23] provides the relax-ordered implementations for the benchmarks falling in first category. The unordered implementation of SSSP, Graph Coloring, PageRank and PageRank-DP are taken from Pannotia [7] and the unordered version of BFS is obtained from Rodinia [24]. 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 the different domains such as road networks, social networks and biological networks. The static input graphs vary from extermely sparse (e.g. USA-CAL) to dense (e.g Twitter) and smaller in size (e.g Cage14) to large (Friendster) .

Different static input graphs along with their vertex count, edge count and average degree are shown in Table 2.3. These static input graphs are converted into temporal graphs through the tool described in Section 2.3. Table 2.3 also shows generated temporal graphs and 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

Table 2.3: 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

accelerator, are broken down into smaller chunks through the Stinger Framework [25] and are processed sequentially. Memory transfer times are not included in the completion times for the 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 to create a fair comparison between different accelerators.

2.6 EVALUATION

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

Table 2.4: Completion time for static and temporal benchmark-input combinations on Intel Xeon 40-core. (SO) is strict-ordered, (RO) is relax-ordered and (UO) is unordered. Completion Times are normalized to their Strict-ordered implementation completion times

Graph	SSSP			BFS			Color			MST		
	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)
CAL	1	**0.81**	0.91	**1**	1.05	1.15	1	**0.3**	0.4	**1**	2.9	2.63
CAL_t_5	1	**0.53**	0.56	1	0.9	**0.8**	1	0.26	**0.24**	**1**	1.8	1.9
CAL_t_10	1	0.38	**0.29**	1	0.53	**0.47**	1	0.19	**0.14**	1	**0.84**	1.2
CAGE	1	**0.25**	0.49	1	**0.52**	0.72	1	**0.4**	0.41	**1**	1.4	3.6
CAGE_t_5	1	0.19	**0.1**	1	0.4	**0.35**	1	0.32	**0.25**	1	**0.95**	2.1
CAGE_t_10	1	0.14	**0.08**	1	0.31	**0.18**	1	0.26	**0.17**	1	**0.72**	1.6
FB	1	**0.16**	0.48	1	**0.56**	0.69	1	**0.57**	0.7	**1**	2.8	5.2
FB_t_5	1	0.14	**0.13**	1	0.45	**0.39**	1	0.50	**0.49**	**1**	1.8	2.2
FB_t_10	1	0.1	**0.09**	1	0.32	**0.25**	1	0.41	**0.3**	1	**0.93**	1.8
Orkt	1	0.78	**0.43**	1	**0.42**	0.45	1	0.91	**0.83**	**1**	1.02	2.3
Orkt_t_5	1	0.57	**0.23**	1	0.35	**0.3**	1	0.7	**0.5**	1	**0.6**	1.2
Orkt_t_10	1	0.41	**0.13**	1	0.21	**0.17**	1	0.43	**0.35**	1	**0.52**	0.74
Twtr	1	1.1	**0.9**	1	1.07	**0.53**	1	0.76	**0.56**	1	**0.4**	0.73
Twtr_t_5	1	0.8	**0.45**	1	0.6	**0.4**	1	0.6	**0.35**	1	**0.31**	0.44
Twtr_t_10	1	0.5	**0.36**	1	0.39	**0.26**	1	0.47	**0.28**	1	0.19	**0.28**

combinations.

Multicore Combinations

Table 2.4 shows the performance variations for all benchmark-inputs combinations (temporal and static) on the Intel Xeon 40-core machine. Completion time in the table are normalized to the completion time of strict-ordered implementation.

Graph benchmarks (SSSP, BFS and Color) with unordered implementations utilizing static distribution of vertices/edges among cores as the parallelization strategy, are highly data parallel. These benchmarks along with large or dense static graphs

Table 2.5: Completion time for static and temporal benchmark-input combinations on NVidia GTX-1080 GPU. (SO) is strict-ordered, (RO) is relax-ordered and (UO) is unordered. Completion Times are normalized to Relax-ordered implementation

Graph	SSSP			BFS			Color			MST		
	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO)	(SO)	(RO)	(UO
CAL	-	**1**	2.2	-	**1**	1.06	-	**1**	1.33	-	**1**	1.49
CAL_t_5	-	**1**	1.01	-	1	**0.6**	-	**1**	1.1	-	1	**0.98**
CAL_t_10	-	1	**0.72**	-	1	**0.38**	-	1	**0.84**	-	1	**0.71**
CAGE	-	1	**0.99**	-	**1**	1.18	-	**1**	1.4	-	**1**	2.0
CAGE_t_5	-	1	**0.74**	-	1	**0.7**	-	1	**0.8**	-	**1**	1.2
CAGE_t_10	-	1	**0.59**	-	1	**0.46**	-	1	**0.51**	-	1	**0.93**
FB	-	**1**	2.87	-	**1**	1.1	-	**1**	1.09	-	**1**	1.56
FB_t_5	-	**1**	1.35	-	1	**0.8**	-	1	**0.55**	-	1	**0.8**
FB_t_10	-	1	**0.92**	-	1	**0.61**	-	1	**0.28**	-	1	**0.67**
Orkt	-	1	**0.94**	-	**1**	1.08	-	**1**	1.03	-	**1**	1.98
Orkt_t_5	-	1	**0.66**	-	1	**0.9**	-	1	**0.52**	-	**1**	1.07
Orkt_t_10	-	1	**0.43**	-	1	**0.79**	-	1	**0.36**	-	**1**	1.02
Twtr	-	1	**0.87**	-	1	**0.93**	-	1	**0.84**	-	**1**	2.1
Twtr_t_5	-	1	**0.52**	-	1	**0.73**	-	1	**0.35**	-	**1**	1.2
Twtr_t_10	-	1	**0.39**	-	1	**0.61**	-	1	**0.55**	-	1	**1.02**

(Orkut, Twitter and Friend) give superior performance for unordered implementation. This is because they have large number of independent tasks that can be executed in parallel with each. In other words, the amount of unordered work in these benchmarks is high due to limited inter-task dependencies and low synchronization. Connected component with all input graphs give superior performance for unordered implementations for the same reason. Unordered implementations are also optimal for the temporal counter-parts of the aforementioned benchmark-input combinations. However, temporal unordered versions show higher performance gains as compare to the static unordered versions for the same combinations. This is because they expose high parallelism as compare 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 to be performed by these workloads is relatively less, leading to limited parallelism. Unordered implementations in these cases perform high redundant work which 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 the 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 implementation 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 with each other 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 require the presence of the task execution order. Hence, relax-ordered implementation gives better performance for temporal SSSP-CAL combination.

Graph benchmarks which 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 counter-parts. Benchmarks such as A* with all input graphs prefer strict-ordered implementation. A* finds a single path from source to 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 prefer strict-ordered implementations for the same reason. However, MST with large or dense graphs (Orkut and Twitter) prefer relax-ordered implementations due to the presence of high reductions. Reductions in the implementation translate to work that is local to work. 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 of time in the relax-ordered implementation to expose more parallelism.

The geometric mean of the optimal implementation's completion time for all for all benchmark-input combinations are calculated using the static choices for temporal combinations and then utilizing true temporal choices for temporal graphs. 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 fort_10 (graphs with 10 times more temporal edges than static) graphs on Intel Xeon-40 core.

This shows that the optimal implementation choices for static graph benchmarks and inputs are not good enough for temporal graph benchmarks and inputs on a multicore as the behvaior of temporal graphs is different from static graphs.

GPU Combinations

Table 2.5 shows the performance variations for all benchmark-inputs combinations (temporal and static) on GTX-1080 GPU. All completion times in the table are normalized to the completion time of relax-ordered implementation. Trends on GPU for the optimal implementations are similar to multicore except that all the benchmark-input combinations that give superior performance for strict-ordered on multicore, map to relax-ordered on GPU.

Static graph benchmarks with high vertex or pareto level parallelism in unordered versions such as SSSP, BFS and Color along with large or dense graphs (Orkut and Twitter) give optimal performance for unordered implementation. 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 order in them. 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,

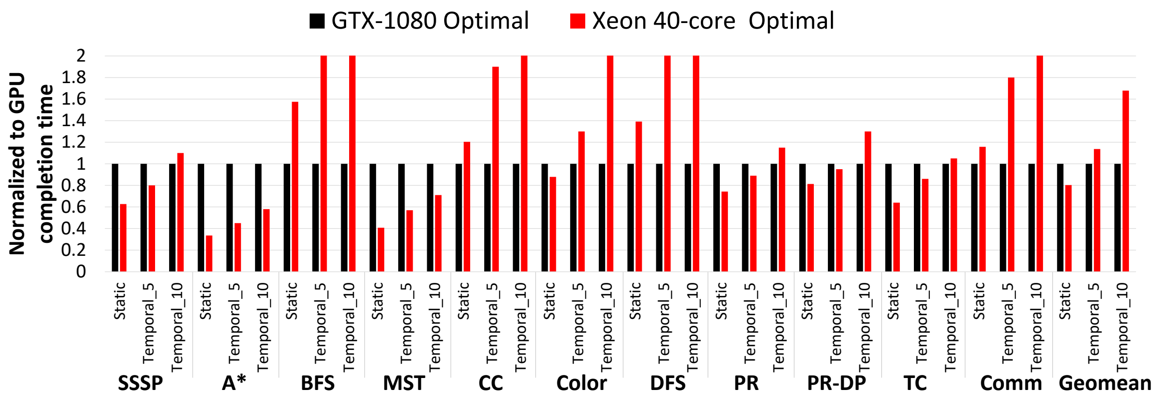

Figure 2.4: Performance comparison of Intel Xeon 40-core and GTX-1080 GPU for all graph benchmarks with average across all static graphs, all temporal_5 graphs and all temporal_10 graphs. Completion Time is normalized to GPU completion time (Higher is worse)

A*, BFS, Color and MST there is a shift in the choice of the optimal implementation from relax-ordered implementation (for statict 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 of the optimal implementation's completion time for all 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. Like multicore, 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.

Inter-Accelerator performance comparison

Figure 2.4 shows the performance comparison of Intel Xeon 40-core machine with NVidia GTX-1080 for different benchmarks. Each benchmark has three completion times 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) for GPU and multicore. All completion times in the figure are normalized to their respective GPU's completion times. It can be observed from the figure that as the input moves 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 GPU due to the presence of the large number of threads. Even for benchmarks such as A* and MST that don't shift to GPU for temporal graphs, display better performance on GPU as compared to their static counter parts.

The static graphs show a geometric performance gain of 24.5% on Intel Xeon 40-core over GTX-1080 GPU, temporal_5 graph show performance gain of 13.74% on GTX-1080 GPU over Intel Xeon 40-core and temporal_10 graph show performance gain of 67.8% on GTX-1080 GPU over Intel Xeon 40-core. This gains show that the accelerator choices for static graph benchmarks and inputs are also not good enough for temporal graphs.

Performance of Time Expanded Graphs

Time Aggregated Graphs are used for the performance comparison of different parallel implementations on multicore and GPU in the preceding sections. This is because performance of different implementations degrades when time expanded graphs are used. This happens as the time expanded graphs perform extremely high redundant

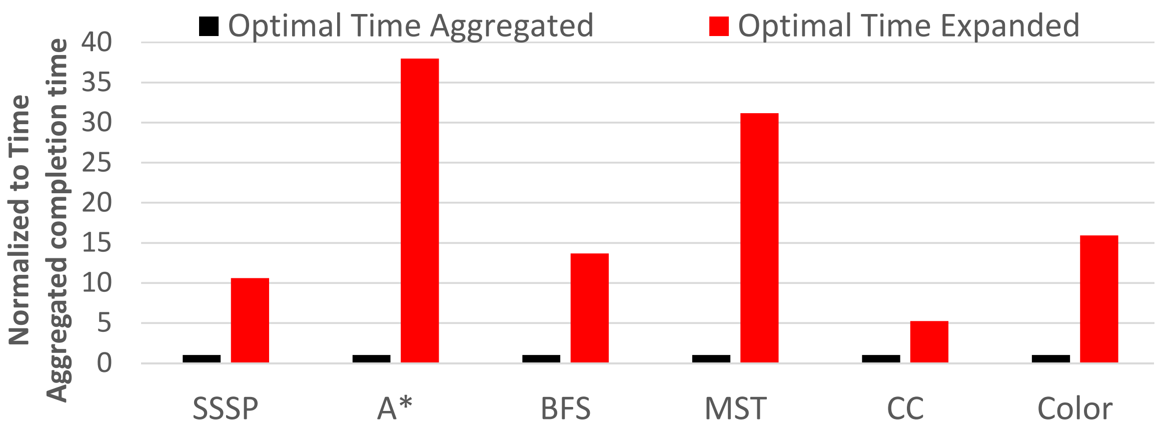

Figure 2.5: Performance comparison of time expanded graphs with time aggregated graphs on Intel Xeon 40-core

work as compare to time aggregated graphs. Figure 2.5 shows the performance comparison of the time aggregated graphs with time expanded graphs on Intel Xeon 40-core. The figure only shows the optimal implementation's completion time averaged (geometric mean) across all input graphs for each benchmark as time aggregated graphs are better for every benchmark-input combination. The completion time is normalized to the completion time of time aggregated graphs. It can be observed that the time-aggregated give better performance than time expanded graphs for all benchmarks. For some benchmarks such as A* and MST, time-expanded graphs perform **30x** worse than time aggregated graphs. This is because these benchmarks expose limited parallelism and redundant work only hurts their performance.

3 RELATED WORK

Prior works in the domain of performance predictor have mainly optimized resource allocation in a single accelerator setup through runtimes of operating system [49] [50] [51].

Graph Analytics and graph input dependence are the key elements that are missing from these works due to enormous search space. There are some other works such as PetaBricks [14, 13] and OpenTuner [12] that optimizes the algorithmic choices within the specific implementation but are indifferent to various implementations of a graph workload. These works suffer from higher computation cost due to the complex algorithmic space. There are some other auto-tuners using regression [52] that are lower in complexity but still are not able to meet real-time requirements. The error rates in the prior predictive performance models are also extremely high for example approximately 27% in [52] due to which performance [53] takes a hit.

There are some works that optimize for the inputs [54] such as XAPP and few others that optimizes the intra-accelerator parameters [55]. HeteroMap [16] is another performance predictor that optimizes for the inter- and intra- accelerator choices for the graph benchmarks and inputs. The proposed work is different from all of the previous performance predictors because it optimizes the inter-implementation choices for the graph benchmarks and inputs. Many real world applications utilize graphs that are time varying in nature. Traffic predictions [29], social networks [30], biological sciences [31] and wireless sensor networks [32] are the few examples where the temporal graphs are used. Representation and modeling of the time varying graphs has been given a lot of attention in the prior literature [34, 33, 35, 36]. The Chronos [56] graph engine proposes a layout scheme for temporal graphs that utilizes the structural and temporal locality to efficiently process these graph. Several conventional graph processing workloads such as shortest path, traversal and spanning tree algorithms are also developed for the temporal graphs [44, 46, 41, 42].

Eventhough there has been lot of research on the representation and algorithmic front, the performance anaylsis of different parallel implementations in terms of optimal implementation for a temporal graph benchmark is not studied in literature. There are three different kinds of parallel implementations present in literature for static graph benchmarks namely strict-ordered [9], relax-ordered [10] and unordered implementations [22]. Like static graphs [13, 54], the performance of an implementation is dependent on the variations in graph benchmarks and inputs. The selection of the optimal implementation is important for temporal graph and benchmarks as it can lead to large performance gains.

4 CONCLUSION

This paper proposes a performance predictor that optimizes the choice of the parallel implementation for a given graph benchmark-input combination on a single machine setup. An analytical prediction model is created that creates a mapping from the characteristics of the benchmark and input to the parallel implementation and accelerator. The evaluation of the predictor shows geometric performance gains of 54% on Intel Xeon E5-2650 v3 40 core machine and 14% on GTX-1080 GPU through the choice of optimal parallel implementation. The parallel implementation choice is also incorporated in the HeteroMap framework to equip it with the capability of the optimal implementation choice. Overall the modified HeteroMap framework has a geometric gain of 31.5% over original HeteroMap's performance. This paper also proposes a conversion tool to transform a static graph into temporal graph by applying various probability distributions to different parts of 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 accelerator. The choice of the optimal implementation leads to the geometric performance gain of 46.38% on Intel Xeon 40-core and 20.30% on NVidia GTX-1080 for the generated temporal graphs. This work also concludes that the optimal implementation for the temporal graphs and their static counter-parts are not always the same, leading to the geometric performance gain of 20% over the same choices for temporal benchmarks and inputs as their static counter-parts.

Bibliography

[1] B. V. Cherkassky, A. V. Goldberg, and T. Radzik, "Shortest paths algorithms: Theory and experimental evaluation," *Mathematical Programming*, vol. 73, no. 2, pp. 129–174, May 1996. [Online]. Available: https://doi.org/10.1007/BF02592101

[2] R. K. Ahuja, *Network Flows: Theory, Algorithms, and Applications*, 1st ed. Pearson Education, 2017.

[3] S. L. Vine, A. Zolfaghari, and J. Polak, "Autonomous cars: The tension between occupant experience and intersection capacity," *Transportation Research Part C: Emerging Technologies*, vol. 52, pp. 1 – 14, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X15000042

[4] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[5] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *WWW '10: Proceedings of the 19th international conference on World wide web.* New York, NY, USA: ACM, 2010, pp. 591–600.

[6] R. S. Xin, D. Crankshaw, A. Dave, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "Graphx: Unifying data-parallel and graph-parallel analytics," *CoRR*, vol. abs/1402.2394, 2014. [Online]. Available: http://arxiv.org/abs/1402.2394

[7] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron, "Pannotia: Understanding irregular gpgpu graph applications," in *2013 IEEE International Symposium on Workload Characterization (IISWC)*, Sep. 2013, pp. 185–195.

[8] F. Busato and N. Bombieri, "An efficient implementation of the bellman-ford algorithm for kepler gpu architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 8, pp. 2222–2233, Aug. 2016.

[9] M. A. Hassaan, D. D. Nguyen, and K. K. Pingali, "Kinetic dependence graphs," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: ACM, 2015, pp. 457–471. [Online]. Available: http://doi.acm.org/10.1145/2694344.2694363

[10] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13. New York, NY, USA: ACM, 2013, pp. 456–471. [Online]. Available: http://doi.acm.org/10.1145/2517349.2522739

[11] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," in *Euro-Par 2015: Parallel Processing*, J. L. Träff, S. Hunold, and F. Versaci, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015, pp. 209–221.

[12] J. Ansel, S. Kamil, K. Veeramachaneni, J. Ragan-Kelley, J. Bosboom, U.-M. O'Reilly, and S. Amarasinghe, "Opentuner: An extensible framework for program autotuning," in *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation*, ser. PACT '14. New York, NY, USA: ACM, 2014, pp. 303–316.

[13] Y. Ding, J. Ansel, K. Veeramachaneni, X. Shen, U.-M. O'Reilly, and S. Amarasinghe, "Autotuning algorithmic choice for input sensitivity," in *The 36th ACM Conference on Programming Language Design and Implementation*, ser. PLDI. New York, NY, USA: ACM, 2015.

[14] J. Ansel and et. al., "Petabricks: A language and compiler for algorithmic choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09. New York, NY, USA: ACM, 2009, pp. 38–49.

[15] P. M. Phothilimthana, J. Ansel, J. Ragan-Kelley, and S. Amarasinghe, "Portable performance on heterogeneous architectures," in *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '13. New York, NY, USA: ACM, 2013, pp. 431–444.

[16] M. Ahmad, H. Dogan, C. J. Michael, and O. Khan, "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, 2019, pp. 268–281.

[17] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015. [Online]. Available: http://networkrepository.com

[18] R. Meusel, S. Vigna, O. Lehmberg, and C. Bizer, "The graph structure in the web analyzed on different aggregation levels," *The Journal of Web Science*, vol. 1, no. 1, pp. 33–47, 2015. [Online]. Available: http://dx.doi.org/10.1561/106.00000003

[19] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[20] S. Beamer, K. Asanovic, and D. Patterson, "Locality exists in graph processing: Workload characterization on an ivy bridge server," in *Int. Symposium on Workload Characterization*, ser. IISWC, 2015.

[21] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan, "Brief announcement: the problem based benchmark suite," in *SPAA*, 2012.

[22] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan, "Crono: A benchmark suite for multithreaded graph algorithms executing on futuristic multicores," in *2015 IEEE International Symposium on Workload Characterization*, Oct 2015, pp. 44–55.

[23] Y. Wang, Y. Pan, A. Davidson, Y. Wu, C. Yang, L. Wang, M. Osama, C. Yuan, W. Liu, A. T. Riffel, and J. D. Owens, "Gunrock: Gpu graph analytics," *ACM Trans. Parallel Comput.*, vol. 4, no. 1, pp. 3:1–3:49, Aug. 2017. [Online]. Available: http://doi.acm.org/10.1145/3108140

[24] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S. Lee, and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, Oct 2009, pp. 44–54.

[25] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, "Stinger: High performance data structure for streaming graphs," in *2012 IEEE Conference on High Performance Extreme Computing*, Sept 2012, pp. 1–5.

[26] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, Eds., *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, November 13-14, 2006*, ser. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, vol. 74. DIMACS/AMS, 2009.

[27] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014.

[28] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, no. 1, pp. 29–123, 2009.

[29] D. Deng, C. Shahabi, U. Demiryurek, L. Zhu, R. Yu, and Y. Liu, "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, 2016, pp. 1525–1534.

[30] N. Santoro, W. Quattrociocchi, P. Flocchini, A. Casteigts, and F. Amblard, "Time-varying graphs and social network analysis: Temporal indicators and metrics," *arXiv preprint arXiv:1102.0629*, 2011.

[31] V. D. Calhoun, R. Miller, G. Pearlson, and T. Adalı, “The chronnectome: time-varying connectivity networks as the next frontier in fmri data discovery,” *Neuron*, vol. 84, no. 2, pp. 262–274, 2014.

[32] S. Lai and B. Ravindran, “On distributed time-dependent shortest paths over duty-cycled wireless sensor networks,” in *2010 Proceedings IEEE INFOCOM*. IEEE, 2010, pp. 1–9.

[33] B. George and S. Shekhar, “Time-aggregated graphs for modeling spatio-temporal networks,” in *Journal on Data Semantics XI*. Springer, 2008, pp. 191–212.

[34] K. Wehmuth, A. Ziviani, and E. Fleury, “A unifying model for representing time-varying graphs,” in *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. IEEE, 2015, pp. 1–10.

[35] V. Nicosia, J. Tang, M. Musolesi, G. Russo, C. Mascolo, and V. Latora, “Components in time-varying graphs,” *Chaos: An interdisciplinary journal of nonlinear science*, vol. 22, no. 2, p. 023101, 2012.

[36] Y. Wang, Y. Yuan, Y. Ma, and G. Wang, “Time-dependent graphs: Definitions, applications, and algorithms,” *Data Science and Engineering*, pp. 1–15, 2019.

[37] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro, “Time-varying graphs and dynamic networks,” in *International Conference on Ad-Hoc Networks and Wireless*. Springer, 2011, pp. 346–359.

[38] J. Tang, S. Scellato, M. Musolesi, C. Mascolo, and V. Latora, "Small-world behavior in time-varying graphs," *Physical Review E*, vol. 81, no. 5, p. 055101, 2010.

[39] J. Byun, S. Woo, and D. Kim, "Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time," *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[40] U. Demiryurek, F. Banaei-Kashani, and C. Shahabi, "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, 2010, pp. 474–477.

[41] H. Wu, J. Cheng, S. Huang, Y. Ke, Y. Lu, and Y. Xu, "Path problems in temporal graphs," *Proceedings of the VLDB Endowment*, vol. 7, no. 9, pp. 721–732, 2014.

[42] H. Wu, J. Cheng, Y. Ke, S. Huang, Y. Huang, and H. Wu, "Efficient algorithms for temporal path computation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 11, pp. 2927–2942, 2016.

[43] B. Ding, J. X. Yu, and L. Qin, "Finding time-dependent shortest paths over large graphs," in *Proceedings of the 11th international conference on Extending database technology: Advances in database technology*. ACM, 2008, pp. 205–216.

[44] S. Huang, J. Cheng, and H. Wu, "Temporal graph traversals: Definitions, algorithms, and applications," *arXiv preprint arXiv:1401.1919*, 2014.

[45] S. Huang, A. W.-C. Fu, and R. Liu, "Minimum spanning trees in temporal graphs," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 2015, pp. 419–430.

[46] W. Hu, H. Zou, and Z. Gong, "Temporal pagerank on social networks," in *International Conference on Web Information Systems Engineering*. Springer, 2015, pp. 262–276.

[47] P. Rozenshtein and A. Gionis, "Temporal pagerank," in *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 2016, pp. 674–689.

[48] J. He and D. Chen, "A fast algorithm for community detection in temporal network," *Physica A: Statistical Mechanics and its Applications*, vol. 429, pp. 87–94, 2015.

[49] S. Panneerselvam and M. Swift, "Rinnegan: Efficient resource use in heterogeneous architectures," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: ACM, 2016, pp. 373–386.

[50] S. Sridharan, G. Gupta, and G. S. Sohi, "Adaptive, efficient, parallel execution of parallel programs," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14. New York, NY, USA: ACM, 2014, pp. 169–180.

[51] H. Pan, B. Hindman, and K. Asanović, "Composing parallel software efficiently with lithe," in *Proceedings of the 31st ACM SIGPLAN Conference on Program-*

ming Language Design and Implementation, ser. PLDI '10. New York, NY, USA: ACM, 2010, pp. 376–387.

[52] N. Ardalani, C. Lestourgeon, K. Sankaralingam, and X. Zhu, "Cross-architecture performance prediction (xapp) using cpu code to predict gpu performance," in *Proceedings of the 48th International Symposium on Microarchitecture*, ser. MICRO-48. New York, NY, USA: ACM, 2015, pp. 725–737.

[53] C. Delimitrou, D. Sanchez, and C. Kozyrakis, "Tarcil: Reconciling scheduling speed and quality in large shared clusters," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*, ser. SoCC '15. New York, NY, USA: ACM, 2015, pp. 97–110.

[54] K. Kofler, I. Grasso, B. Cosenza, and T. Fahringer, "An automatic input-sensitive approach for heterogeneous task partitioning," in *Proceedings of the Int. Conference on Supercomputing*, ser. ICS '13. New York, NY, USA: ACM, 2013, pp. 149–160.

[55] M. Ahmad, C. J. Michael, and O. Khan, "Efficient situational scheduling of graph workloads on single-chip multicores and gpus," *IEEE Micro*, vol. 37, no. 1, pp. 30–40, Jan 2017.

[56] W. Han, Y. Miao, K. Li, M. Wu, F. Yang, L. Zhou, V. Prabhakaran, W. Chen, and E. Chen, "Chronos: a graph engine for temporal graph analysis," in *Proceedings of the Ninth European Conference on Computer Systems.* ACM, 2014, p. 1.