Accelerating Relax-ordered Task-parallel Workloads using Multi-Level Dependency Checking

Masab Ahmad*
University of Connecticut
masab.ahmad@uconn.edu

Mohsin Shan*
University of Connecticut
mohsin.shan@uconn.edu

Akif Rehman
University of Connecticut
akif.rehman@uconn.edu

Omer Khan
University of Connecticut
khan@uconn.edu

ABSTRACT

Work-efficient task-parallel algorithms enforce ordered execution of tasks using priority schedulers. These algorithms suffer from limited parallelism due to data movement and synchronization bottlenecks. State-of-the-art priority schedulers relax the ordering of tasks to avoid false dependencies generated by strict queuing constraints, thus unlocking task parallelism. However, relaxing task dependencies results in shared data races among cores that lead to redundant task computations in concurrently executing threads. Although static algorithm optimizations have been shown to reduce redundant work, they do not exploit the tradeoff between parallelism and work efficiency that is only exposed during runtime. This paper proposes a task dependency checking mechanism that dynamically tracks the monotonic property of parent-child relationships across multiple levels from any given task. Since shared memory writes are known to be slower than concurrent reads, the multi-level checks effectively detect task dependency races to prune redundant tasks. Evaluation of relax-ordered algorithms on a 40-core Intel Xeon multicore shows an average of 44% performance improvement over the Galois *obim* scheduler.

CCS CONCEPTS

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

KEYWORDS

Task-parallelism, Task Scheduling, Ordered Algorithms, Concurrency, Consistency, Shared Memory, Multicore.

ACM Reference Format:
Masab Ahmad, Mohsin Shan, Akif Rehman, and Omer Khan. 2020. Accelerating Relax-ordered Task-parallel Workloads using Multi-Level Dependency Checking. In *2020 International Conference on Supercomputing (ICS '20), June 29-July 2, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3392717.3392758

*Both authors contributed equally to this research.

ICS '20, June 29-July 2, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7983-0/20/06…$15.00
https://doi.org/10.1145/3392717.3392758

1 INTRODUCTION

Task-parallel algorithms represent a growing area of activity in high performance computing (HPC) [30], especially in single-node embedded settings [10]. For example, graph processing has emerged as an important application domain for supercomputing. Graph algorithms achieve work-efficiency by executing tasks in the order executed by their sequential counterparts. As graph vertices (parents) create tasks for their connected edges (children), ordering between tasks is implemented by tracking the read-write dependencies on preceding or future tasks with parent-child relationships. The execution order between tasks is then enforced using queuing primitives that impose fine grain communication and synchronization between cores [16, 24]. This limits performance scaling on shared memory parallel machines with large core counts. Prior literature has proposed relax-ordered and unordered parallel graph algorithms that break consistency guarantees between tasks [23]. While this allows for more parallelism, it results in work inefficiency since the algorithm requires multiple iterations of redundant task processing to converge on a solution.

The Galois priority scheduler [23] enables methods to process unordered and relax-ordered algorithms on shared memory parallel machines. For unordered implementations, Galois does not impose any priority scheduling of tasks within and across cores. However, it enforces local ordering of tasks on a per-core granularity for a relax-ordered implementation. Inter task dependencies are tracked using monotonic updates to their shared data values during the execution of an algorithm. An example can be taken from the shortest path problem, where shared data values of vertices start off as a large number, and then lower down once optimal paths are found from a source vertex to each other vertex. The *monotonically* decreasing property to reach convergence for a vertex enables a relax-ordered implementation to redundantly execute that task in different cores, while the end result is guaranteed to contain the lowest distance from the source vertex. In general, a task-parallel algorithm with a monotonically decreasing or increasing shared data values for tasks enables priority schedulers to implement a relax-ordered variant of an ordered algorithm. This introduces a tradeoff between between the amount redundant computations and the exploitable parallelism exposed by a relax-ordered algorithm.

An efficient relax-ordered algorithm strives to minimize its redundant task computations, while maximizing the exploitable parallelism. Prior works, such as KLA [14] prune redundant work using static algorithmic mechanisms. The Δ-stepping algorithm for shortest path computations is one such example algorithm.

However, static measures require costly off-line pre-processing to optimize pruning parameters, or have sub-optimal on-line performance due to dynamically changing input diversity. Pre-processing is not entirely applicable on graphs that are being streamed in as chunks or as sub-graphs, as certain HPC settings require real-time constraints [25]. As static measures do not fully exploit the work-efficiency and parallelism tradeoff, a dynamic work pruning mechanism is desirable.

This paper proposes a multi-level task dependency checking mechanism for relax-ordered algorithms that efficiently prunes out redundant work at runtime. It is well known that shared memory writes are slower than concurrent reads. Therefore, updates of shared data values propagate across cores with a significant delay, while the concurrent reads of per-core task data structures proceed and result in redundant processing of tasks. This challenge can be solved by either making races in shared memory faster, or by doing multi-level parent–child dependency checking to prune tasks whose parents have been overwritten by new data values. These checks exploit the *monotonic property* of shared data values, where an algorithm's converging shared data values either only increase or decrease.

As the monotonic property implies that an algorithm's order may be relaxed, the proposed multi-level checking scheme is applicable on all relax-ordered implementations. When a monotonic shared data value is updated by a task, it may violate global order as its latest counterpart task may have been executed in another core. A check across multiple parent–child levels detects and prunes this task from being processed redundantly. This detection is done using shared memory reads of the latest parent values, and comparing them to the old values that were stored alongside the task at insertion into the per-core ordering data structure. The multi-level task dependency check is then performed based on the monotonic property of a given algorithm. The proposed check is implemented on top of the Galois priority scheduler, and evaluated on a 40-core Intel Xeon multicore, as well as 72-core Tilera multicore machine. Results for a range of relax-ordered algorithms show an average of 44% performance improvement for the 40-core Xeon over the Galois relax-ordered implementations.

2 BACKGROUND AND MOTIVATION

In task-parallel programming paradigm, a task is the basic unit of computation that executes in parallel with other tasks. Tasks may have dependencies on each other via parent–child relationships, or they may operate on shared data leading to intra task dependencies. Many task-parallel algorithms impose an order on processing tasks for work efficient execution [24]. This can be strictly the order in which sequential versions execute tasks (ordered), or tasks execute in each core with certain relaxed ordering constraints (relax-ordered). These ordering variants introduce tradeoffs in work-efficiency and parallelism. Strictly ordered algorithms are highly work-efficient, but suffer from a lack of parallelism due to communication and synchronization bottlenecks. However, when ordering is relaxed, tasks may take multiple iterations to converge. This leads to redundant task computations that degrade work efficiency but expose more parallelism.

Algorithm 1 Generic Task-Parallel Algorithm Pseudocode

```
Global List (L), L.push (source_task).      ▷ Global List for ordered only
Local Queue (Q), Q.push (source_task).
Core ID (tid).
1:  while (Q ≠ ∅) in each tid do
2:      parent = Q.top ()          ▷ top() with priority in ordered and relax
3:      test = safe_src_test(parent)          ▷ safe_src_test for ordered
4:      if (test == pass) then
5:          parent = Q.pop ()  ▷ pop() with priority in ordered and relax
6:          L.atomic_remove ()                    ▷ L.remove () in ordered
7:          for (each child of parent) do
8:              Critical Section Task Work ()
9:              Q.push (child)      ▷ Priority Order for ordered and relax
10:             L.atomic_add (child)                  ▷ L.add () in ordered
```

2.1 Ordered vs. Relax-Ordered Execution

Ordered algorithms require a strict execution order of tasks among cores, where inter task relationships are tracked using shared data structures. Kinetic Dependence Graph (KDG) [16] is a state-of-the-art task-parallel framework that allows parallel execution of tasks with strict ordering using per-core priority scheduling, and a global task ordering list. When a task is dequeued in a core, a *safe-source test* is applied to check if that task has a dependency on another task in the system. If a dependency is detected, the task execution is stalled until other tasks with higher priority execute. Otherwise, the task executes and potentially creates new children tasks that are scheduled for subsequent processing. Algorithm 1 shows a generic per-core pseudocode for a task-parallel algorithm. A *parent* task is first looked-up with priority from the per-core local queue, Q (line 2). It is then checked for ordered execution using *safe-source test* on the global list, L (lines 3-4). When the test passes, *parent* is dequeued from Q and L (lines 5-6), and performs its task executor functions (lines 7-8). The *child* tasks with parent–child relationships are then pushed to Q and L (lines 8-9). Parallelizing KDG ordered implementation is challenging because the global list L requires frequent atomic synchronizations.

In a relax-ordered implementation, global task ordering is relaxed to allow parallel execution of created *child* tasks. However, tasks still rely on shared data updates to track their progress towards a termination condition. Since data consistency on global task ordering is weakened, the same task may execute multiple times in different cores to achieve convergence. For relax-ordered, the Algorithm 1 is modified to remove the safe-source test and the global list, L (remove lines 2-4, 6, and 10). However, the per-core queue (Q) still operates with priority scheduling. Galois *obim* [23] is a state-of-the-art priority scheduler that implements relax-ordered algorithms to minimize communication, and expose parallelism among tasks.

It is possible to further relax the local per-core task execution to expose more parallelism. In such unordered algorithms, the local queue, Q is replaced with a simple array, and tasks are processed without priority. Here, tasks iteratively execute until the algorithmic convergence is achieved, which greatly degrades work efficiency. Unordered algorithms are useful when the exploitable parallelism delivers more benefits than the redundant computations needed for convergence.

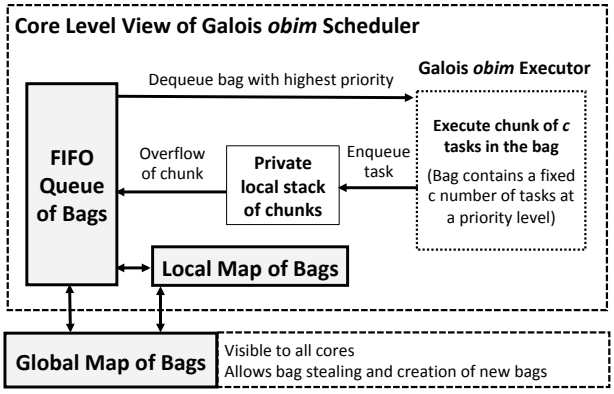

Figure 1: Data structures and their interactions in the Galois *obim* task scheduler.

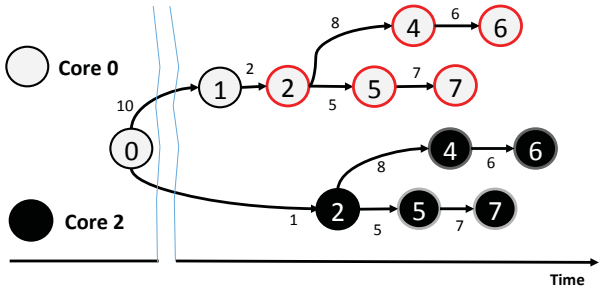

Figure 2: Single source shortest path (SSSP) algorithm example using the Galois *obim* scheduler.

2.2 The Galois *obim* Scheduler

Galois ordered by integer metric (*obim*) [23] is a state-of-the-art relax-ordered priority scheduler. It is a distributed design that aims to mitigate communication and synchronization costs between cores, and improve the work-efficiency of task-parallel algorithms. Figure 1 shows the core level view of various data structures and their interactions for *obim*. It implements processing of tasks by associating a task priority level to a data structure, called *bag*. Each bag contains multiple tasks with the same priorities, and is thus allowed to execute without any order within a core. Another local data structure, *FIFO queue of bags* maintains the list of bags a core is allocated to process. In collaboration with the *Local Map of bags*, the core dequeues the highest priority bag, and starts executing all tasks one by one from that bag. During the execution of a task, new children tasks belonging to different priority levels are created. These children tasks are enqueued in a per-core local data structure, called *Private local stack of chunks*. In *obim*, each core contains a data structure, called a *chunk* that is a ring-buffer with 8 to 64 tasks (fixed at compile time). When a chunk gets full, it is enqueued into the *FIFO queue of bags*. This step requires that the chunk becomes visible to all other cores in the system. As mentioned earlier, a core dequeues a bag from the *FIFO queue of bags*. If the queue is empty, the core steals a bag from another core's *FIFO queue of bags*. A global data structure, *Global Map of bags* maintains the list of all bags in the system. The *Local Map of bags* is just a cache of this global map. Thus, when a local map is updated when a chunk is inserted into the *FIFO queue of bags*, it must synchronize with the global map. Moreover, when a core dequeues from an empty *FIFO queue of bags*, it refreshes the local map with information in the global map, and tries again.

The *obim* scheduler performs certain tasks multiple times in different cores due to lack of strict synchronization between cores. Cores only periodically synchronize with each other when bags are moved between cores. Moreover, tasks within a bag are processed by a core without any ordering constraint. Each core tracks inter-task dependencies using a unique timestamp per task, which is used as a metric of convergence. The timestamp values are checked when a child task is created by a high priority task in a core. However, an update to a shared data timestamp value may not propagate before a concurrent read by a a different core. This results in the relax-ordered algorithms to miss detecting these race conditions on timestamp values. As a consequence, children tasks are created that lead to future redundant work for the system to process.

Figure 2 depicts a representative flow for the single source shortest path (SSSP) algorithm using the *obim* scheduler. All initial timestamps in shared memory have very large or infinite values, after which tasks update their values starting from core 0, executing task 0. The color shading represents the core processing a task, while the number represents the task ID being processed. Each edge represents its distance value from a source to a destination task. At some point in time, task ID 2 is dequeued in both cores 0 and 2. However, the timestamp value being updated for task ID 2 in core 0 is 12, while in core 2 is 1. Due to read/write race conditions, the lower timestamp in core 2 does not propagate to core 0, which ends up creating children tasks 4 and 5. These unnecessary tasks create their own paths, leading to redundant work being created and processed by core 0. The red circles are shown to represent tasks that are processed redundantly in this example. The objective of this paper is to track and detect redundant tasks as early as possible, and mitigate unnecessary work performed by relax-ordered algorithms.

2.3 Main Idea of Pruning Redundant Tasks

This paper proposes a method to prune redundant tasks from processing. It takes advantage of the monotonic property and shared memory concurrent reads of task timestamps. For monotonically decreasing timestamps, only the smallest timestamp value updated by any core is visible in shared memory. The *obim* priority scheduler only checks the global timestamp value for the task that is being created. However, the parent of this task may have already propagated its latest timestamp value in shared memory. If a check can be performed on this parent task that reveals that it has a lower timestamp in shared memory compared to its timestamp at the time of its creation, the parent task need not proceed to execute. In other words, the check on the parent task reveals that some other core has processed the same task ID with a lower timestamp, and thus the current task is being performed redundantly in the system. This check can be expanded to include the parent's parent task, or even higher levels of parent–child relationships, thus creating the notion

of *multi-level checking* to prune redundant paths. On one hand, a deeper level check enables early detection of redundant paths, thus improving work efficiency. On the other hand, the multi-level checks add additional data access and logical comparison overheads that may not always be useful. The idea of the proposed multi-level checking can be visualized using the SSSP example from Figure 2. It is evident that both cores 0 and 2 are processing task ID 2. However, the update to core 2 has a smaller timestamp of 1 that does not propagate to core 0 when it processes tasks 2 with a timestamp of 12. Since the race between cores 0 and 2 for update on task 2's timestamps are never detected, core 0 is able to create redundant children tasks 4 and 5, which subsequently create more redundant tasks 6 and 7. The proposed 1-level check will make sure that tasks 4 and 5 in core 0 check their parent (task 2) for its timestamp values. As core 2 would have updated task 2's timestamp in shared memory at the time of these 1-level checks, the tasks 4 and 5 are pruned out from further processing in core 0.

3 TASK-PARALLEL EXECUTION & MULTI-LEVEL DEPENDENCY CHECKING

The Galois *obim* scheduler relaxes task order by ensuring local per core ordering of *bags*. This improves parallelism as global synchronization of bags between cores is minimized. A shared data structure on a per task granularity is maintained in global memory, which temporally tracks timestamps. A task updates this shared timestamp array to track its progress towards algorithmic convergence. For example, in the SSSP case it is a shared distance array that is atomically updated when a task is executed. As the distance from the source vertex ID always decreases for convergence, this distance array is updated to monotonically decrease. Other target task-parallel algorithms may implement the shared timestamp structure to monotonically increase or decrease. In Galois *obim* scheduler, a check is performed at the creation of each task to ensure that only children tasks that have the potential to improve the algorithm's convergence are processed. The proposed *multi-level* checking increases the scope of this scheduler to prune redundant tasks. For example, in a 1-level check, the parent task of the task(s) being created is checked for the timestamp value at the time of its creation against its latest shared timestamp array value. If the monotonic property being tracked clears this task for further processing, then the children task(s) are allowed to proceed. Otherwise, the path is terminated at this parent task. A 2-level or even deeper *n*-level checks are also possible by incorporating the relevant timestamp accesses, as well as comparisons for each parent task being tracked.

Algorithm 2 shows the pseudocode for a relax-ordered algorithm with the proposed *multi-level* checks incorporated on top of the Galois *obim* scheduler. Each core executes a function called an `Executor()` that operates on the per-core queue, *Q[tid]*. In the baseline Galois version, lines 4–5 and 11–17 are not executed. The baseline Galois version first dequeues a task (T_p) from *Q[tid]*, which is then executed to create children tasks using the for loop pseudocode on lines 6–10. The algorithm specific critical section check on the timestamp values of a child task determines whether to proceed with the creation of this task. Here, the child task's current timestamp value is compared against the latest corresponding

Algorithm 2 Relax-Ordered Algorithm Pseudocode with Proposed Multi-Level Checking

Input: source task s, input data.
Output: Timestamps in D[] from s.
$D[] \leftarrow \infty$, $D_s \leftarrow 0$ ▷ Monotonically Decreasing Example
$p \leftarrow$ parent, $c \leftarrow$ child, $barr \leftarrow$ GlobalBarrier $C_w \leftarrow$ Worker Core
$tid \leftarrow$ CoreID, $T \leftarrow$ Tasks, $T_n \leftarrow$ Tasks from n level work.
$Q[tid] \leftarrow$ Queue for each tid

```
1:  procedure EXECUTOR(Q[tid], T_n)
2:      while (Q[tid] ≠ ∅) do
3:          T_p = Q[tid].popTask                    ▷ Pop T_p Parent Task
4:          test = SPECCHECKTASK(T_p)
5:          if (test == pass) then
6:              for (each child c of parent p) do
7:                  Critical Section Checks, and Work on D[]
8:                  COMMITTASK(D[c])
9:                  T_c = CREATETASK(c)
10:                 Q[tid].pushTask(T_c)
11: procedure SPECCHECKTASK((T_p)) return p
12:     p = pass
13:     for L = 1, L < n levels do              ▷ Start from current Task
14:         p = Task(p.L)                       ▷ Get Task id from Level
15:         if (D[p] < p.oldtimestmp) then
16:             KILLTASK(T_p)                               ▷ Kill T_p
17:             p = fail
18: procedure CREATETASK((c)) return T_c
19:     T_c =TASK(c, c.p, c.oldtimestmp, c.p.oldtimestmp)
```

timestamp value in shared memory array ($D[]$). Only a child task that passes this check is processed by the `CommitTask(D[c])` function, and pushed into *Q[tid]*. In the baseline Galois version, the `CreateTask(c)` function only needs to track the task ID (c) and current timestamp ($c.oldtimestamp$) value of the child task T_c that is being created.

The proposed multi-level checking includes lines 4–5 in Algorithm 2. As with the baseline Galois version, the `Executor()` operates on the per-core queue, *Q[tid]*. However, each dequeued task (T_p) calls the `SpecCheckTask()` function to invoke the proposed multi-level checks. The depth of a multi-level checking is controlled by the L parameter. For example, $L = 1$ only checks for the current task T_p, while $L = 2$ invokes a second check for the parent task ID of T_p as well. The loop on lines 13–17 goes through all configured levels, and checks the corresponding task's timestamp value when it was inserted in its bag against the latest timestamp value in shared memory array ($D[]$). This check is done only for task T_p, when $L = 1$. However, for $L = 2$, the parent task of T_p is also checked in addition to the task T_p. The specific check depends on the monotonic property. For monotonically decreasing case, a task is deemed redundant (and killed using `KillTask()` function) if the multi-level check determines that a timestamp value in shared memory is lower than that task's timestamp value at its time of creation. `SpecCheckTask()` returns a value p to `Executor()` to specify whether to proceed with T_p, or not. For the algorithms evaluated in this paper, `KillTask()` disallows execution of the current task, T_p, and the core moves on to the next task in $Q[tid]$. If the check passes (line 5), task T_p executes and creates children task(s) following the same procedures discussed for the baseline Galois

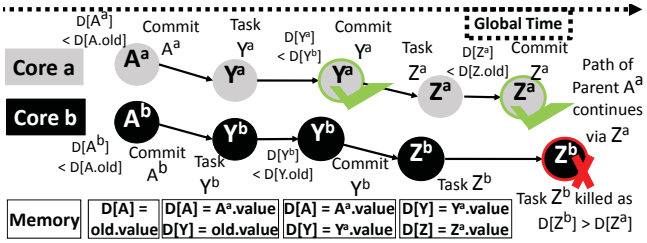

Figure 3: 1-level dependency checking scenario to detect redundant path in the relax-ordered SSSP benchmark.

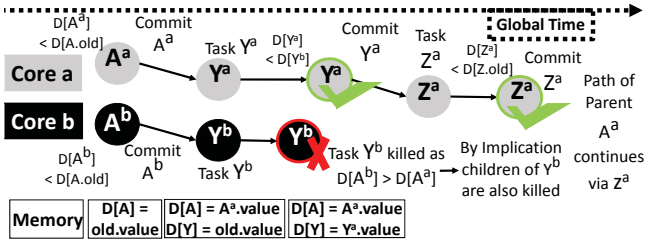

Figure 4: 2-level dependency checking scenario to detect redundant path in the relax-ordered SSSP benchmark.

version. However, the metadata for each child task T_c is modified on line 19 to include its parent's task ID ($c.p$) and the timestamp ($c.p.oldtimestamp$). The parent's timestamp is the timestamp of T_p when it was first pushed into $Q[tid]$. This metadata is used for the multi-level checks on lines 14–15.

3.1 Checking and Pruning Redundant Tasks

This section describes the proposed multi-level checking in Algorithm 2 using a representative execution on two cores for the single source shortest path (SSSP) graph algorithm.

1-Level Dependency Checking: Figure 3 shows a 1-level check in SSSP, where the shared timestamp array ($D[]$) track's the monotonically decreasing timestamp values at the per task granularity. Tasks A^a and A^b execute in Cores a and b at the same time. Assume that both tasks read the old value of A in $D[A]$ to execute and push their children tasks, Y^a and Y^b to their respective core's queue. At the end of this commit, the final value in the $D[A]$ is that of A^a, since $D[A^a] < D[A^b]$. In this case, if A^b updates first, then A^a replaces it as it has a smaller timestamp, ultimately storing $D[A^a]$ in $D[A]$. However, since both A^a and A^b are committed, their children tasks, Y^a and Y^b are both pushed into their respective core's queue. Task Y^b sees the old value of Y in $D[Y]$, and commits Y^b to create Z^b. Task Y^a in Core a sees the value of Y^b in $D[Y]$, but updates $D[Y]$ with the timestamp of Y^a as $D[Y^a] < D[Y^b]$, while also creating child task Z^a. Now, Core b slows down due to certain microarchitectural variations, and the two cores interleave such that Core a extracts the task Z_a first, and commits its value to $D[Z]$. At a later global time, Core b extracts task Z^b and the check (line 15) on it fails as $D[Z^b] > D[Z^a]$, where $D[Z^a]$ is already stored in memory, thereby eliminating the redundant path started by A^b. In this case, a chain of redundant tasks initiated by A^b are not killed until task Z^b's termination. With a multi-level check on the parent of a task, it is possible to detect that timestamp values have updated to kill redundant paths earlier in their execution. This strategy is discussed next.

***n*-Level Dependency Checking:** It is possible that the parent or even the parent of the parent of the current task is redundant. This implies that at the time of checking a task ID, the deeper level tasks in the parent–child relationship may already have been updated with a timestamp, such that the path can be classified as redundant. To accomplish this deeper n-level check, a task's parent's ID and timestamp is also pushed in the queue, as shown on line 19 of Algorithm 2. Moreover, lines 11–17 show how such checks are performed across L levels. While deeper checks may improve work-efficiency by classifying redundant paths earlier in their execution, they also add the overhead and checking of stored parent metadata. Thus, an n-level check trades off work-efficiency with the acquired performance.

Figure 4 shows a 2-level checking for the SSSP algorithm. Utilizing the same scenario from Figure 3, Cores a and b go down the same path, while reading stale timestamp values in shared memory array ($D[]$). Task A^b enqueues task Y^b, while parent A^a enqueues child task Y^a, to their respective core's queue. By the time Core b extracts task Y^b, the timestamp value of A^a has made it to shared memory, and is visible to all cores. In the 2-level check, Core b checks the timestamp of Y^b, and finds that it is the lowest timestamp ($D[Y^a]$ has not propagated to memory yet). Core b also checks if the parent's timestamp has changed since it was enqueued. Task Y^b was enqueued with a parent timestamp of $D[A^b]$, but it has been updated with a lower timestamp in $D[A]$. Thus, task Y^b is killed as its parent is redundant due to another core updating $D[A]$ with a lower timestamp than $D[A^b]$. In the 1-level check, Core b went down a redundant path as seen in Figure 3, only to have the path killed after executing several redundant tasks. With the 2-level check, Core b detects the redundant path earlier in the execution, thus improving overall performance.

3.2 Heuristic for Multi-Level Checking

By default, this paper performs a 2-level check for all evaluated benchmarks, unless otherwise stated. This means that from any source, checks occur on the task itself, and its parent. Deeper checks may also be made, such as checking the parent of the parent of the current task. In doing so, the parent ID and it's level must be encoded in every task, along with the timestamp with which it pushed its children into the queue. For each additional level, this increases memory requirements by two integer values per task for singular timestamps. To study this tradeoff, this paper performs a sensitivity study in the evaluation section on how many levels may suffice for best performance.

It is empirically observed that 2-level checks induce performance overheads for dense graphs. This happens because dense graphs are observed to not induce long redundant paths. As tasks are killed earlier using 1-level check due to shallow redundant paths, a check on the parent task (2-level) adds unnecessary computations in the

scenario when a path is not redundant. Thus, as an optimization, the master core tracks the input density [5, 11], and applies the default 2-level check for sparse inputs, and 1-level check for dense inputs. This is similar to the distinction done in [10], where applying sparse or dense edge mapping classification allows reduced edge checking. In this heuristic, an input is labeled as sufficiently dense if at least P (P=total cores) tasks are distributed among worker cores in the first two levels.

3.3 Overheads of Multi-Level Checking

Multi-level checking adds storage overheads of two integer values per task per level, where these values include the parent ID, and the parent timestamp with which the task is created. However, these values are encoded in the task's `struct` to preserve locality. Once a task is loaded in a core, the check results in a branch and a comparison, after which the core either executes the task, or does not execute it (kills it). As out-of-order cores efficiently hide latency, the checking latency at a given level is not expected to incur significant overhead. However, for deeper level checks, the data access and checking overheads start to offset performance gains achieved via work-efficiency. In terms of programmability, not much overhead is required since the integer values required for checking are incorporated in the input (graph) structure, and checking functions are made as library calls.

4 TARGET TASK-PARALLEL BENCHMARKS

Relax-ordered task parallel algorithms from Galois [23] utilize the *obim* priority scheduler described in Section 2.2. If a benchmark or machine does not have a Galois implementation with the *obim* scheduler (e.g. Color), then the Galois Lonestar [18] version is ported. If any Galois version is not present (e.g. A*), then a state-of-the-art relax-ordered algorithm is ported from literature. For a GPU comparison, all benchmarks are acquired from the Gunrock [34] benchmark suite. The only exception is the A* benchmark, which is not ported for GPU comparisons. All benchmark inputs use the compressed sparse row (CSR) format, while all shared memory structures use atomic operations for updates.

Single Source Shortest Path (SSSP) algorithm finds shortest paths from a source to all vertices in a graph. A distance array in shared memory, $D[]$ maintains the distances for each vertex from the source vertex. These distance values are initialized at a large number, and as tasks execute they are lowered monotonically. Therefore, timestamps in SSSP monotonically decrease in the distance array.

The sequential implementation is acquired from Dijkstra's algorithm with C++ priority queue primitives from the Boost library [2], while the parallel ordered version is acquired from KDG [16]. The unordered Δ-stepping algorithm is acquired from state-of-the-art Julienne-Ligra suite [29].

A* Shortest Paths (A*) utilizes a heuristic to prune the work done in traditional SSSP by not visiting all vertices of the input graph. The heuristic distances from the source vertex are also tracked using a monotonically decreasing distance array, $D[]$. Due to a significant reduction in the number of tasks processed, A* is known to be notoriously difficult to parallelize.

The parallel ordered implementation uses a single A* search to a destination vertex, where various random destination vertices are selected and the average completion time is reported across all considered destination vertices [25]. This implementation also uses a priority queue primitive to process tasks, similar to the SSSP counterpart. The unordered implementation spawns multiple A* paths across threads, and the one with the shortest distance to a destination vertex is selected [35].

Breadth-First Search (BFS) starts from a source vertex, and searches vertices in a graph using the edge first method [7]. As edges are searched, the distance of the search increases from the source vertex. This distance increases monotonically, and tracked using a shared distance array, $D[]$.

The sequential version uses a queue and an array to specify whether a vertex is searched or not, and is acquired from C++ Boost library [2]. The parallel ordered implementation is acquired from KDG [16]. The unordered version of BFS opens and parallelizes pareto fronts, and it is acquired from the CRONO benchmark suite [4].

Minimum Spanning Tree (MST) implements the Prim's algorithm, which uses a priority queue and key checks based on the input graph to update critical sections [7]. The checks on keys decrease monotonically, which are tracked using the shared array, $Key[]$. However, the parents of any given vertex are stored in a $parent[]$ array.

The sequential version is acquired from the C++ Boost library [2], and uses Krustal's algorithm (same complexity as Prim's), while the parallel ordered version is acquired from KDG [16]. The unordered version also uses Krustal's algorithm, where the outer loop is parallelized among vertices, and is acquired from the problem based benchmark suite (PBBS) [28].

Connected Components (CC) labels edges to a component in an input graph. Components of each vertex increase monotonically, and are tracked using the shared array, $CC[]$.

The sequential ordered version uses a depth-first search using a queue to label components, while the parallel ordered implementation parallelizes queue updates [7]. The unordered version uses the Shiloach-Vishkin algorithm from the CRONO benchmark suite [4].

Graph Coloring (Color) implements vertex coloring based on their saturation degree. Vertex colors increase monotonically, and are tracked using the shared array, $Color[]$.

The sequential version is acquired from the C++ Boost library [2], while the parallel ordered version is acquired from [1]. The unordered version is acquired from Pannotia [6], which is converted to execute on a multicore CPU.

5 METHODOLOGY

5.1 Experimental Setup

CPU machines with large core counts are used to evaluate the proposed multi-level checking scheme integrated with state-of-the-art Galois *obim* priority scheduler. The primary goal of the evaluation is to show how representative task-parallel algorithms improve parallelism and performance scalability for different machines. Thus, performance comparisons to a GPU are also evaluated.

Table 1: Input Graph Datasets.

Graph	$\|V\|$	$\|E\|$	Size(GB)
USA-Cal(CAL) [9]	1,890,815	4,657,742	0.1
com-Orkut(OR) [26]	3,072,627	234,370,166	4
Twitter(Twtr) [19]	41,652,231	1,468,365,182	27
Friendster(Frnd) [22]	124,836,180	3,612,134,270	52
M.Br.Ret.3(CO) [21]	562	577,350	0.05
Cage14(Cage) [26]	1,505,785	27,130,349	0.1

The **Intel Xeon E5-2650 v3** multicore CPU with Turbo Boost capability is the primary machine used for evaluation. It has 10 hyper-threaded cores executing at 3.00GHz in each of its 4 sockets. Moreover, the machine has 1TB DDR4 RAM, and a 25MB L3 last-level cache. All benchmarks use the `pthread` library, and are compiled using the g++ compiler (v 6.4.1) with the -O3 optimization flag on the 40-core Intel CPU.

The **Tilera *Tile-Gx72™*** [8] multicore processor is utilized to further validate the performance gains achieved by the proposed multi-level checking in priority task schedulers. This machine implements 72 cores on a single die, with hardware directory based cache coherence to support efficient shared memory paradigm. Each core integrates three execution pipelines, a two-level private–shared cache hierarchy, and five mesh interconnection networks. The machine executes at 1.2GHz with a total of 23.5MB on-chip cache capacity, and 110$Tbps$ on-chip communication bandwidth. Four 72-bit DDR3 controllers with ECC support 16GB memory capacity. The benchmarks use the `pthread` library, and compiled using g++ (v 5.4.1). Furthermore, the Galois *obim* scheduler is ported using Tilera specific libraries and APIs.

The performance gain from the 40-core Intel CPU are compared against state-of-the-art GPU implementations of the target benchmarks. The **NVidia GTX 1080** GPU is utilized, which has 2560 CUDA cores executing at 1.73GHz, and a 8GB GDDR5X main memory. All programs executing on the GPU use the `OpenCL` library.

5.2 Benchmark Inputs

Several diverse directed graphs are chosen to show that the proposed multi-level checking scheme performs well on various graph characteristics. Table 1 shows the evaluated graphs and their characteristics. These graphs represent varying degree, diameter, sparsity, and sizes. All evaluated graphs fit in the memory of the 40-core Intel machine. However, the memory size of the Tilera and GPU machine require some graphs (such as Twitter) to be processed using a Stinger-like framework [12, 27]. Stinger processes graphs in chunks (4GB chunk sizes in our case), allowing streaming of edges and graph information into the target processor. While reads are done locally on the current available chunk, writes on vertices not existing in currently available chunk are saved for later when the chunk is available.

For the SSSP benchmark using Δ-stepping algorithm, the optimized Δ values are as follows: 50000 for CAL, 16384 for OR, 32768 for Twtr and Frnd, 4 for CO, and 16 for the CAGE graph.

5.3 Evaluation Metric

Completion times are measured only for the time spent in the parallel region of each benchmark. All pre-processing times to calculate required parameters are added to the overall completion time. For streaming graphs, time spent in disk accesses is not added to the completion time. The Stinger framework used for streaming graphs overlaps latency when moving graph chunks between machine memory and disk. Disk latencies are expected to be oblivious to the time spent in DRAM accesses from the multicore's standpoint.

Table 2: Completion times (in seconds) for Ordered, Unordered, and Relax-ordered algorithms with Galois and the proposed Multi-Level Checking on the 40-core Intel CPU.

	Input	Ordered	Unordered	Galois Relax-Ord	**Galois+Check Proposed**
SSSP	CAL	1.09	0.91	0.31	**0.04**
	OR	1.19	0.24	0.29	**0.10**
	TWTR	7.91	5.52	6.19	**2.51**
	FRND	17.4	6.98	9.10	**3.78**
	CO	0.05	0.02	**0.01**	0.01
	CAGE	0.10	0.04	0.03	**0.01**
A*	CAL	0.23	0.57	0.38	**0.46**
	OR	0.20	0.33	0.41	**0.16**
	TWTR	4.11	185	9.43	**3.99**
	FRND	15.9	85	29.3	**13.5**
	CO	**0.02**	0.05	0.03	0.03
	CAGE	**0.10**	0.21	0.18	0.12
BFS	CAL	0.46	0.61	0.28	**0.11**
	OR	0.81	0.79	0.45	**0.20**
	TWTR	12.2	7.43	8.89	**7.23**
	FRND	19.4	16.5	14.3	**8.97**
	CO	0.03	**0.02**	0.04	0.04
	CAGE	0.15	0.12	0.08	**0.04**
MST	CAL	0.11	0.18	0.08	**0.03**
	OR	0.41	0.55	**0.28**	0.29
	TWTR	13	9.53	8.16	**6.31**
	FRND	39.7	18.4	26	**12.5**
	CO	0.04	**0.02**	0.04	0.03
	CAGE	**0.09**	0.12	0.16	0.11
CC	CAL	0.06	0.02	0.03	**0.02**
	OR	0.18	0.12	0.14	**0.11**
	TWTR	15.8	**7.89**	9.13	8.05
	FRND	19.6	**10.4**	12.6	11.3
	CO	0.05	**0.02**	0.03	0.04
	CAGE	0.25	0.11	0.10	**0.08**
Color	CAL	0.27	0.09	0.12	**0.05**
	OR	0.12	0.10	0.11	**0.09**
	TWTR	11.3	8.61	6.33	**5.65**
	FRND	19.2	12.3	16.4	**10.6**
	CO	0.03	0.01	**0.01**	0.02
	CAGE	0.22	0.09	0.08	**0.05**

6 EVALUATION

6.1 Performance and Task Parallelism

Table 2 shows the completion times (in seconds) for all algorithm implementations and input graphs for the 40-core Intel CPU machine. The details of various benchmark implementations are outlined in Section 4. The relax-ordered implementation uses the Galois *obim* priority scheduler ported on the evaluated 40-core machine. The *proposed* column in the table presents the evaluation of the

multi-level checking that is implemented on top of the Galois *obim* relax-ordered benchmarks. The heuristic from Section 3.2 is utilized, which selects 1-level checking for input graphs CO and CAGE, while the remaining inputs use the default 2-level checking.

For SSSP, BFS, MST and Color benchmarks, the relax ordered implementations under the Galois *obim* scheduler perform better than both ordered and unordered counterparts. Moreover, unordered versions expose sufficient task-level parallelism that the 40-core machine exploits better than overcoming the synchronization overheads of ordered task processing in ordered variants. The Galois scheduler exploits inter task locality by processing tasks together in bags with clustered timestamp ranges. This specifically helps graphs with higher density (e.g., CO and CAGE), as many children tasks are generated by a parent task with little timestamp divergence. The Galois scheduler also processes bags with priority ordering, thus reducing redundant tasks as compared to the unordered variants. The proposed multi-level checking performs better or at par with the relax-ordered Galois baseline. These benchmarks exhibit high dependence on ordering levels, as timestamp values propagate across tasks. As redundant task updates lead to race conditions across cores, the multi-level checking enables a mechanism to prune such paths earlier in their execution. This helps input graphs that exhibit higher levels of task dependencies, such as the CAL graph. However, as graphs become dense and task timestamp divergence drops (CO and CAGE), the overheads of multi-level checking cannot be overcome by the benefits from pruning redundant tasks. Thus, such graphs perform at par with the baseline Galois scheduler.

The A* benchmark is notorious to parallelize due to its singular shortest path search towards a goal vertex using a heuristic that visits a limited number of vertices at each level. This makes redundant task processing difficult to exploit parallelism along the paths that lead to algorithmic convergence. Consequently, A* performs better with the KDG ordered implementation as compared to both unordered and relax-ordered Galois implementations. The proposed multi-level checking coupled with Galois' locality and priority aware task scheduling leads to competitive performance compared to KDG ordered implementation. The multi-level checks prune a significant number of redundant tasks. Moreover, the additional checks are less costly compared to fine grain synchronizations required in the ordered implementation. Overall, the A* benchmark is observed to outperform with the proposed scheme for graphs that exhibit higher levels of task dependencies across parent–child relationships. As discussed earlier, dense graphs show less path divergence, and hence the proposed multi-level checking overheads are hard to overcome as compared to the ordered implementation.

Connected components (CC) is a highly parallel benchmark that benefits significantly from unordered processing of tasks. As tasks perform significant work in between shared timestamp updates, the dependencies propagate seamlessly in the unordered implementation. Consequently, the unordered algorithm does not increase the number of redundant tasks over the KDG ordered counterpart, and thus delivers high performance. Both Galois and the proposed multi-level checks add some overheads while only pruning a small number of redundant tasks. Therefore, the completion times for CC are observed to be at par with the unordered counterpart. Overall,

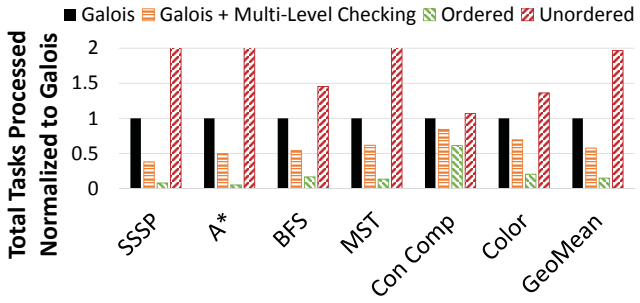

Figure 5: Total tasks executed per benchmark normalized to the Galois baseline on the 40-core setup.

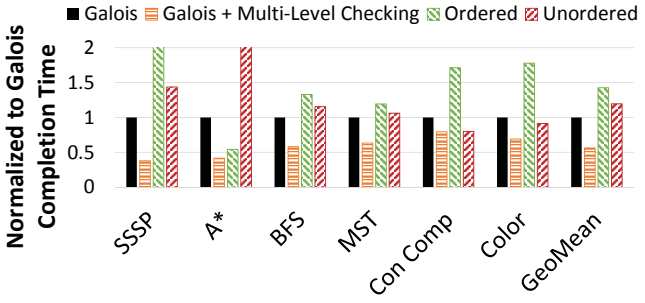

Figure 6: Completion time per benchmark normalized to the Galois baseline on the 40-core setup.

the Galois *obim* scheduler outperforms both ordered and unordered implementations. Moreover, the proposed multi-level checking improves performance of Galois *obim* implementations by 44%.

6.1.1 Work-Efficiency vs. Performance Tradeoff: Figure 5 shows the total tasks processed per benchmark averaged across all input graphs. The ordered, unordered, and relax-ordered Galois and proposed multi-level implementations are normalized to the Galois baseline. Total tasks processed are acquired by counting the number of tasks dequeued and executed to produce children task(s). The unordered implementation processes significantly more tasks compared to all other variants. The only exception is the CC benchmark, which does not process many redundant tasks as it efficiently resolves inter-task dependencies between cores. The KDG ordered implementation performs the least amount of redundant tasks since it synchronizes task ordering between cores to execute tasks with the same complexity as the sequential algorithm counterpart. The proposed multi-level checks prune a significant number of tasks compared to both unordered and relax-ordered implementations. On average, a 2× reduction in tasks processed is observed compared to the Galois baseline. This translates to performance benefits, as observed in Figure 6, which shows normalized completion times averaged over input graphs relative to the Galois baseline.

6.2 Sensitivity to Number of Levels Checked

The performance gains from the proposed multi-level checking depend on the number of parent–child task relationships that are

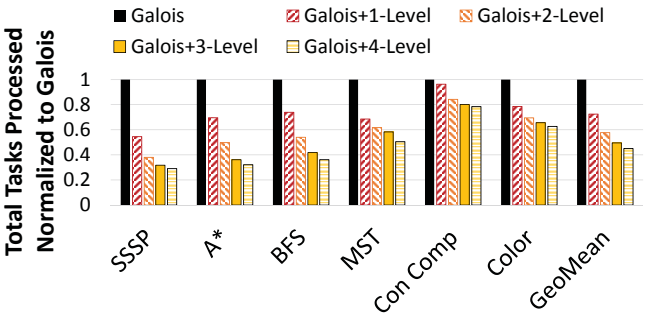

Figure 7: Total tasks executed for various multi-level checks normalized to the Galois baseline on the 40-core setup.

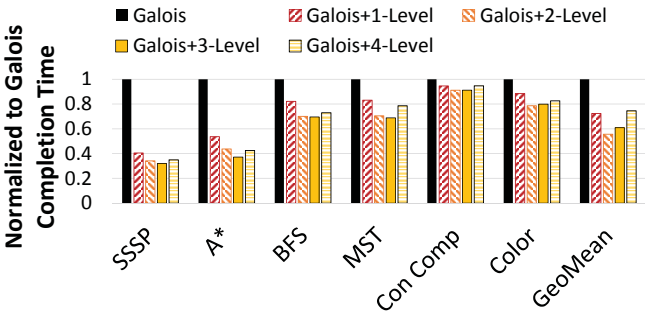

Figure 8: Completion times for various multi-level checks normalized to the Galois baseline on the 40-core setup.

checked to determine whether a task needs to be pruned or not. The higher the number of levels that are checked, the more computations and data accesses are performed on the timestamp values. However, the cross-core race conditions that arise from shared timestamp updates are detected faster. Thus, a higher number of redundant tasks are detected with the higher number of levels that are checked.

Figure 7 shows the total tasks processed for all benchmarks averaged across all inputs, normalized to the Galois baseline. The number of levels checked with the proposed multi-level scheme are varied from 1 to 4 levels. The 1-level check prunes redundant tasks in all benchmarks, except CC. As discussed earlier, CC executes core-private computations in between shared timestamp checks, thus inter-core task dependencies resolve faster in this benchmark and 1-level check is not as useful. However, it is observed that a 2-level check prunes ~20% of the processed tasks in CC. Among the other benchmarks, the most number of redundant tasks are pruned in the SSSP benchmark, followed by A*, BFS, MST and Color, respectively. The 1-level check prunes a geometric mean of 25% tasks, while the 2-level check prunes >40% tasks compared to the Galois baseline. As the number of levels increase, all benchmarks show a drop in the number of tasks processed compared to the 1-level check. However, this drop is more prominent from 1-level to 2-level, while deeper level checks yield lower detection of redundant tasks. This indicates that deeper level checks look for inter-core data races that have already resolved by the time their shared timestamp data reads are performed.

Table 3: Study of multi-level checking on the 40-core CPU. Geometric mean speedups over sequential implementation reported across benchmarks on a per input granularity.

Checking Levels	CAL	OR	Twtr	Frnd	CO	CAGE
No Level Check	2.7	4.4	8.7	9.3	3.8	7.8
1-level Check	5.8	8.2	14	14	**5.1**	**11**
2-level Check	6.6	**8.6**	**20**	**16**	4.8	10
3-level Check	**6.7**	8.1	16	13	3.1	7
4-level Check	5.6	5.9	9.3	8.8	1.7	2.3

Although deeper level checks are clearly beneficial in reducing the amount of redundant tasks processed, they accompany increasing number of computations and data accesses. These overheads are incurred for all tasks, thus their performance impact on non-redundant tasks must be overcome by the benefits of pruning redundant tasks. Figure 8 shows the normalized completion time of 1-level to 4-level checking for all benchmarks normalized to the Galois baseline. Due to the tradeoff between tasks pruned and overheads of multi-level checking, the performance benefits are observed to maximize at 2-level checking on the 40-core Intel CPU machine. The trends from the number of tasks processed in Figure 7 are also observed for the performance gains. The CC benchmark shows the least performance benefits over the Galois baseline, while the SSSP benchmark gains most advantage from the proposed multi-level checking.

6.3 Sensitivity to Benchmark Inputs

Table 3 shows the impact of multi-level dependency checks for the 40-core CPU setup using performance speed-ups over the sequential implementations. The performance gains are averaged across all benchmarks, and reported for each input graph. The *No Level Check* version only perform inter-task dependency checks for the child task being created. However, the 1-level check also incorporates the task being processed, while higher level checks involve the parent(s) of the task being processed. The inputs that perform well with higher level checks imply that deeper levels of inter-task dependency chains are being updated, leading to faster detection of redundant tasks. The CAL input has a high diameter, which concurs with this argument and perform best at 3-level checking. However, graphs with low diameter and high edge density (such as CO and CAGE) work best with 1-level checking due to lack of dependency chains between parent–child task relationships. The proposed heuristic from Section 3.2 selects 1-level checking for CO and CAGE inputs, while all other inputs are executed with the default 2-level check. Although CAL input slightly outperforms with 3-level checking, the heuristic's choice of 2-level check performs within 2% performance variation.

6.4 Sensitivity to Parallel Machine Type

A multicore processor resolves shared data races depending on the implementation of the on-chip core pipelines, caches, networks, and coherency and consistency protocols. To evaluate the efficacy of

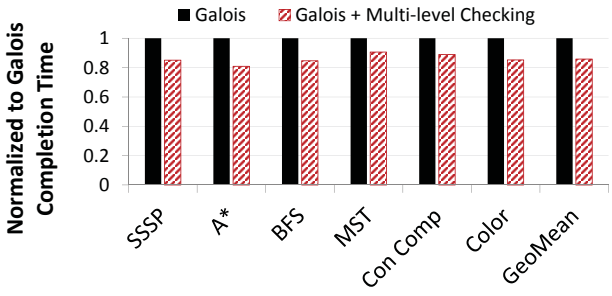

Figure 9: Completion time per benchmark normalized to the Galois baseline on the Tilera multicore machine.

the proposed multi-level checking on a different multicore machine type, a 72-core Tilera single-chip processor is evaluated. Figure 9 shows the completion time of the multi-level checking scheme for all relax-ordered benchmarks normalized to the Galois *obim* baseline. The geometric mean performance gains of 18% are observed for the Tilera multicore, which is lower compared to the 40-core Intel machine. These performance gains increase to ∼30% (data not shown) for the CAL graph with deeper inter-task dependency chains. However, the performance gains drop to ∼12% for the dense CAGE graph that has shallow inter-task dependencies.

The Tilera machine implements a highly scalable directory-based hardware cache coherence protocol that supports fast shared memory updates between cores. Moreover, the 72 cores of the Tilera machine are all integrated on a single 2-D mesh interconnection network with low per-hop latency and high bandwidth for core-to-core communication. The Tilera machine also implements weaker cores compared to the Intel Xeon server class out-of-order cores. Since a Tilera core cannot hide long latency stalls as efficiently as a Xeon core, the computations and data accesses in between shared timestamp updates incur more latency. Consequently, the Tilera multicore resolves inter-task dependencies faster, and computations between timestamp updates allow enough time for the resolved dependencies to propagate between cores. Thus, the benefits from the proposed multi-level checks are lower in Tilera compared to the Intel Xeon multicore.

6.5 GPU Comparison

GPUs expose massive hardware threading capabilities that can be exploited to hide the latency of redundant computations in relax-ordered or even unordered task-parallel algorithms. Therefore, a large NVidia GPU with orders of magnitude large number of cores and very high main memory bandwidth capability is compared against the proposed multi-level checking scheme executing on the 40-core Intel CPU machine. The GPU executes state-of-the-art benchmark implementations from the Gunrock suite [34]. Table 4 shows the geometric mean completion times for all benchmarks averaged across the evaluated inputs. The CC benchmark is a highly parallel algorithm with a small portion of its tasks redundantly processed during execution (c.f. Figure 5). Therefore, the GPU outperforms the CPU implementation of the CC benchmark by 40%. However, for all other benchmarks, the CPU consistently outperforms the GPU with performance benefits ranging from 6% to 36%.

Table 4: Completion times shown per benchmark. GPU Gunrock implementations are compared against the proposed multi-level checking on the 40-core Intel CPU.

	SSSP	A*	BFS	MST	CC	Color	Geo
40-core CPU	**0.34**	0.66	**0.61**	**0.52**	0.35	**0.34**	**0.45**
GPU–Gunrock	0.53	-	0.65	0.77	**0.21**	0.53	0.49

It is also noteworthy that Gunrock does not implement a GPU benchmark for the A* algorithm. This is primarily due to the lack of exploitable parallelism in such a task-parallel algorithm, making it unsuitable for GPU adoption.

7 RELATED WORK

KDG [16], which is a successor of the Galois priority scheduler [23] exploits parallelism in ordered task-parallel algorithms. It implements per-core queues that execute ordered task processing within and across cores. In KDG, after popping a task from a core's priority queue, it is first checked for its global priority order before processing. The *safe-source test* at task issue time checks if any other core is executing the same task ID with a higher priority timestamp value. This check is enforced using blocking synchronization between all cores. This stalls cores from doing work, thus creating inter-thread contention and limiting parallelism. However, optimal work-efficiency is ensured as no redundant tasks are executed compared to the sequential algorithm counterpart.

Certain ordered algorithms, such as shortest-path Δ-stepping [29], are relaxed by statically identifying tasks that can be executed without strict ordering constraints. Other works, such as KLA [14] relax order by statically determining the number of steps required for write-based synchronizations. However, such works do so in a static fashion, relying on statically available information to calculate Δ parameter values that are used to identify tasks that can execute with relaxed ordering [33]. The Galois framework [23] utilizes an ordered by integer metric (*obim*) runtime scheduler that relaxes ordering for task execution. In this paper, we focus on a dynamic approach to identify and prune redundant tasks on top of the Galois *obim* scheduler [20]. Here, the relaxed ordering is also dynamically controlled by allowing tasks to drift from their strict priority order. However, a novel multi-level parent–child task dependency checking mechanisms is proposed to ensure redundant tasks are detected early during their execution. The proposed relax-ordered implementation avoids unnecessary work to balance the tradeoff between work-efficiency and performance.

Swarm [17] and related works [3, 32] build on KDG and TLS schemes [31] to maintain priority queues in hardware, and accelerate task-parallel ordered algorithms using speculative task execution. Due to synchronization and strict consistency requirements in multiple parent–child levels, KDG only achieves speedups similar to prior TLS schemes [13, 15], while Swarm achieves many-fold speedups. Swarm tracks 8 levels of parent-child relationships in hardware by allowing speculation on tasks within these levels, and also by doing task commits and kills in a single cycle. This means that it can keep track of 8 levels of vertex-edge relationships in hardware, and commit or kill (with rollback) work within a few cycles. In contrast, KDG checks tasks globally at issue time, which

eliminates the need for task aborts and kills, but reduces parallelism. On the other hand, Swarm checks tasks at commit time, and requires hardware support for task level abort, rollback, memory logging, and kill mechanisms. Swarm also requires changes to the hardware cache coherence protocol, requires support to rollback the globally propagated state, and implements an arbiter to globally order tasks across cores. In comparison, the proposed multi-level checking based priority scheduler is a general purpose software scheme that executes on commercially available shared memory multicore machines with no complex hardware modifications.

8 CONCLUSION

Due to various existing ordered and unordered implementations to solve task-parallel algorithms, significant performance variations are observed for general purpose parallel machines. To improve work-efficiency and parallelism, prior works relax the order in which tasks execute by exploiting monotonic property of shared data value updates in such algorithms. However, state-of-the-art relax-ordered algorithms do not fully exploit work-efficiency due to the addition of redundant work for correctness, and by only applying static checks for work pruning. Taking these issues in context, this work proposes runtime multi-level dependency checks that exploit the underlying algorithm's monotonic parent-child relationships to unlock parallelism and improve work-efficiency. The evaluation on a 40-core Intel Xeon multicore CPU shows that the proposed task scheduler improves performance by an average of 44% over state-of-the-art relax-ordered algorithms executing on the Galois *obim* priority scheduler.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Grant CNS-1718481. This research was also funded by the U.S. Government under a grant by the Naval Research Laboratory.

REFERENCES

[1] 2014. Coloring Complex Networks. http://algo2.iti.kit.edu/schulz/abschluss/thesis_huebner.pdf
[2] 2015. Boost C++ Libraries. http://www.boost.org/
[3] M. Abeydeera, S. Subramanian, M. C. Jeffrey, J. Emer, and D. Sanchez. 2017. SAM: Optimizing Multithreaded Cores for Speculative Parallelism. In *26th International Conference on Parallel Architectures and Compilation Techniques*. 64–78.
[4] M. Ahmad, F. Hijaz, Q. Shi, and O. Khan. 2015. CRONO: A Benchmark Suite for Multithreaded Graph Algorithms Executing on Futuristic Multicores. In *IEEE International Symposium on Workload Characterization*.
[5] M. Ahmad and O. Khan. 2016. GPU concurrency choices in graph analytics. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10. https://doi.org/10.1109/IISWC.2016.7581278
[6] S. Che, B. M. Beckmann, S. K. Reinhardt, and K. Skadron. 2013. Pannotia: Understanding irregular GPGPU graph applications. In *IEEE International Symposium on Workload Characterization (IISWC)*. https://doi.org/10.1109/IISWC.2013.6704684
[7] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press.
[8] Tilera Corporation. 2014. TILE-Gx72 Processor. (2014). http://www.mellanox.com/related-docs/prod_multi_core/PB_TILE-Gx72.pdf
[9] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson (Eds.). 2009. *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, 2006*. DIMACS/AMS.
[10] Laxman Dhulipala, Guy E. Blelloch, and Julian Shun. 2018. Theoretically Efficient Parallel Graph Algorithms Can Be Fast and Scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures* (Vienna, Austria) *(SPAA)*. 393–404.
[11] Yufei Ding, Jason Ansel, Kalyan Veeramachaneni, Xipeng Shen, Una-May O'Reilly, and Saman Amarasinghe. 2015. Autotuning Algorithmic Choice for Input Sensitivity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. 379–390.
[12] D. Ediger, R. McColl, J. Riedy, and D. A. Bader. 2012. STINGER: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*. 1–5. https://doi.org/10.1109/HPEC.2012.6408680
[13] Alvaro Estebanez, Diego R. Llanos, and Arturo Gonzalez-Escribano. 2016. A Survey on Thread-Level Speculation Techniques. *ACM Comput. Surv.* 49, 2, Article 22 (June 2016), 39 pages. https://doi.org/10.1145/2938369
[14] Harshvardhan, Adam Fidel, Nancy M. Amato, and Lawrence Rauchwerger. 2014. KLA: A New Algorithmic Paradigm for Parallel Graph Computations. In *ACM International Conference on Parallel Architectures and Compilation* (Edmonton, AB, Canada) *(PACT 2014)*. 27–38.
[15] Muhammad Amber Hassaan. 2016. *Parallelization of ordered irregular algorithms*. PhD Thesis, UT Austin.
[16] Muhammad Amber Hassaan, Donald D. Nguyen, and Keshav K. Pingali. 2015. Kinetic Dependence Graphs. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. 15.
[17] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez. 2015. A scalable architecture for ordered parallelism. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 228–241.
[18] Milind Kulkarni, Martin Burtscher, Calin Casçaval, and Keshav Pingali. 2009. Lonestar: A Suite of Parallel Irregular Programs. In *ISPASS '09: IEEE International Symposium on Performance Analysis of Systems and Software* (Boston, MA, USA). http://iss.ices.utexas.edu/Publications/Papers/ispass2009.pdf
[19] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a social network or a news media?. In *WWW '10: Proceedings of the 19th international conference on World wide web* (Raleigh, North Carolina). 591–600.
[20] Andrew Lenharth, Donald Nguyen, and Keshav Pingali. 2015. Priority Queues Are Not Good Concurrent Priority Schedulers. In *Euro-Par 2015: Parallel Processing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 209–221.
[21] J. W. Lichtman, H. Pfister, and N. Shavit. 2014. The big data challenges of connectomics. In *Nature Neuroscience 17*.
[22] Robert Meusel, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizer. 2015. The Graph Structure in the Web – Analyzed on Different Aggregation Levels. *The Journal of Web Science* 1, 1 (2015), 33–47. https://doi.org/10.1561/106.00000003
[23] Donald Nguyen, Andrew Lenharth, and Keshav Pingali. 2013. A Lightweight Infrastructure for Graph Analytics. In *ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. NY, USA, 16.
[24] Keshav Pingali and et. al. 2011. Ordered vs. Unordered: A Comparison of Parallelism and Work-efficiency in Irregular Algorithms. In *ACM Symposium on Principles and Practices of Parallel Programming* (San Antonio) *(PPoPP)*. 10.
[25] Luis Henrique Oliveira Rios and Luiz Chaimowicz. 2010. A Survey and Classification of A* Based Best-First Heuristic Search Algorithms. In *Advances in Artificial Intelligence – SBIA 2010*, Antônio Carlos da Rocha Costa, Rosa Maria Vicari, and Flavio Tonidandel (Eds.). Springer Berlin Heidelberg, 253–262.
[26] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on AI*.
[27] Mo Sha, Yuchen Li, Bingsheng He, and Kian-Lee Tan. 2017. Accelerating Dynamic Graph Analytics on GPUs. *Proc. VLDB Endow.* 11, 1, 107–120.
[28] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, Phillip B. Gibbons, Aapo Kyrola, Harsha Vardhan Simhadri, and Kanat Tangwongsan. 2012. Brief announcement: the problem based benchmark suite. In *SPAA*.
[29] Julian Shun and et. al. 2017. Julienne: A Framework for Parallel Graph Algorithms Using Work-efficient Bucketing. In *Proc. of the 29th ACM SPAA* (Washington, DC, USA). 12.
[30] George M. Slota, Jonathan W. Berry, Simon D. Hammond, Stephen L. Olivier, Cynthia A. Phillips, and Sivasankaran Rajamanickam. 2019. Scalable Generation of Graphs for Benchmarking HPC Community-Detection Algorithms. In *ACM International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC 2019)*. New York, NY, Article Article 73, 14 pages. https://doi.org/10.1145/3295500.3356206
[31] J. Greggory Steffan, Christopher B. Colohan, Antonia Zhai, and Todd C. Mowry. 2000. A Scalable Approach to Thread-level Speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (Vancouver, British Columbia, Canada) *(ISCA '00)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/339647.339650
[32] S. Subramanian, M. C. Jeffrey, M. Abeydeera, H. R. Lee, V. A. Ying, J. Emer, and D. Sanchez. 2017. Fractal: An execution model for fine-grain nested speculative parallelism. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 587–599. https://doi.org/10.1145/3079856.3080218
[33] Guozhang Wang, Wenlei Xie, Alan Demers, and Johannes Gehrke. CIDR 2013. Asynchronous Large-Scale Graph Processing Made Easy.
[34] Yangzihao Wang, Yuechao Pan, Andrew Davidson, Yuduo Wu, Carl Yang, Leyuan Wang, Muhammad Osama, Chenshan Yuan, Weitang Liu, Andy T. Riffel, and John D. Owens. 2017. Gunrock: GPU Graph Analytics. *ACM Trans. Parallel Comput.* 4, 1, Article 3 (Aug. 2017), 49 pages. https://doi.org/10.1145/3108140
[35] Yichao Zhou and Jianyang Zeng. 2015. Massively Parallel A* Search on a GPU. In *AAAI*.