HD-CPS: Hardware-assisted Drift-aware Concurrent Priority Scheduler for Shared Memory Multicores

Mohsin Shan
University of Connecticut, Storrs, CT USA
mohsin.shan@uconn.edu

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

***Abstract*—Efficiently exploiting parallelism remains a challenging problem in multicore processors. For many algorithms, executing tasks in some priority order results in a work-efficient execution. However, searching high-priority tasks requires communication that hampers performance. A concurrent priority scheduler (CPS) selects high-priority tasks and schedules them on different cores. Modern CPS designs offer various strategies to select high-priority tasks at low communication cost for improved performance. However, they do not explicitly track the priority of tasks and cannot adjust task distribution if the cores are processing low-priority tasks. Moreover, they cannot estimate the right amount of communication required to select high-priority tasks. This paper critically observes that the cores' priority drift can be quantified and used for better performance. A novel CPS design, HD-CPS is proposed to use priority as a signal to optimize drift and communication at runtime. Furthermore, compute-intensive task transfer and processing aspects of the CPS are offloaded to per-core local hardware at a low cost to enhance performance. HD-CPS is shown to consistently improve performance over several state-of-the-art software-based and hardware-assisted CPS designs. With hardware-assist, it approaches near-linear performance scaling as a function of core counts for large shared-memory multicores.**

***Keywords*-Task-Level Parallelism, Concurrent Priority Scheduler, Shared-Memory Multicore, Hardware Messages**

I. Introduction

Graph algorithms are universally used in domains like robotics [1], web search [2], and data mining [3]. This ubiquitous nature has led to various methods to execute them on shared memory multicores for enhanced performance [4]. Concurrent Priority Schedulers (CPS) [5], [6], [7], [8] are proposed to execute these algorithms concurrently. CPS is a data structure that stores newly created tasks and distributes them among cores for processing. CPS generally utilizes a task-based paradigm, decomposing an algorithm into tasks during runtime. Each task is associated with a graph node, and processing it requires several instructions. The cores process tasks and generate new tasks at runtime for processing. When a core generates a new task T, an algorithm-defined priority P is associated with it.

Prior works (e.g., [9]) have shown that task-parallel graph algorithms benefit from processing tasks roughly according to their priority order. However, following priority order poses several challenges to a CPS design. The cores need to communicate to select high-priority tasks, which hampers performance. A CPS can mitigate the communication burden by selecting sub-priority tasks. However, processing low-priority tasks can lead to the generation of further low-priority tasks. This process worsens work efficiency, which is the total number of tasks processed compared to the sequential algorithm [10]. On the other hand, a CPS strives to select high-priority tasks, leading to high communication effort among the cores [11], [12]. Therefore, a good CPS needs to select high-priority tasks for processing and low communication cost to be effective.

RELD [10] is a distributed CPS design that uses a priority queue per core, where cores continuously distribute tasks for processing. This continuous distribution leads to load-balanced execution in the multicore, but incurs high communication cost for task transfers. On the other hand, OBIM [9] uses a global work-list for cores to fetch high-priority tasks. The global work-list is shared by all cores, thus allowing faster propagation of tasks among the cores. However, the global work-list requires high synchronization among the cores. OBIM mitigates communication overheads by processing tasks in bulk using the idea of a bag of tasks. It merges tasks with similar priority range into bags of tasks, and executes one bag at a time in a core. The challenge with this approach is to ensure high bag utilization that requires manual tuning for bag size, and the range of task priorities. PMOD [10] introduces a heuristic to tune bag size at runtime. Both OBIM and PMOD serialize task transfers and processing, thus incurring communication costs that hamper performance. Minnow [13] introduces the idea of decoupling task transfers from task processing. Building on top of OBIM, it implements a separate helper core to prefetch bags, while each worker core performs task processing for the current bag. Using this additional hardware cost, Minnow extracts high task-level parallelism.

All works mentioned above do not *explicitly* track whether cores execute high-priority tasks. We observe that the lack of tracking task priorities leads to redundant computations and unnecessary communication across cores. We introduce *priority drift* as the average difference of priority between the global highest priority task and the tasks being processed

2378-203X/22/$31.00 ©2022 IEEE
DOI 10.1109/HPCA53966.2022.00046

by the cores at a given execution instance. A CPS can measure this metric at different time epoch-based intervals. Priority drift is an important signal as it directly correlates to work efficiency and communication cost. A low priority drift among the cores implies that most cores are processing high-priority tasks, which improves work efficiency. Moreover, priority drift depends on the flow of high-priority tasks among the cores, making it an excellent proxy to track and optimize communication cost.

We propose HD-CPS, a software-hardware CPS design that co-optimizes priority drift and communication cost at runtime. HD-CPS decouples task transfer and processing at the software level, which prevents serialization between these two phases. It implements a per-core software receive queue for inter-core task transfers, allowing task processing to overlap transfer of tasks. The idea of bags from OBIM is adopted to reduce the number of priority queue (PQ) operations. A runtime heuristic optimizes bag utilization to ensure that the benefits obtained by using bags outweigh the computation costs of creating bags. HD-CPS proposes a software feedback-driven runtime heuristic that calculates priority drift among cores at different coarse-grain time intervals. The priority drift at each interval is compared against the previous intervals' priority drift, and the task distribution method adapts at runtime to optimize work efficiency and communication cost.

The task transfers and PQ operations are identified as two significant time-consuming software overheads. However, these operations can be accelerated by offloading them to hardware. HD-CPS introduces a lightweight per-core hardware receive queue and a priority queue that does not require global intervention. These queues offer lower latency than their software counterparts, and are co-designed with software to ensure optimized hardware overhead. The hardware queues further improve priority drift by accelerating task transfer and processing, resulting in high performance scaling. With hardware support, HD-CPS outperforms Minnow despite using considerably less hardware.

HD-CPS and state-of-the-art CPS designs, OBIM, PMOD, Minnow, and RELD are evaluated using an Intel Xeon machine with 40 cores. A software variant of Minnow is modeled on the Intel machine by allocating dedicated cores as minnow cores. The evaluation is performed using representative graph benchmarks and inputs. HD-CPS is shown to improve performance by 1.25× and 1.12× respectively against PMOD and Software Minnow. The hardware enhancements to HD-CPS are implemented in a RISC-V based multicore simulator. HD-CPS improves performance over Minnow with dedicated minnow cores by 8%.

II. Related Work and Motivation

Task-based parallel programming models have gained popularity because they are general-purpose and have superior performance [10]. This paradigm dynamically decomposes an algorithm into tasks that are scheduled to different cores for parallel processing. In a graph setting, each parent task represents a node that performs operations and creates new children tasks (or nodes). Each task has a priority associated with it, which depends on the algorithm.

There are two ways to process tasks according to their priority. An *unordered* execution disregards priority, and the algorithm executes tasks in an arbitrary order. This execution mode results in ample parallelism but requires additional iterations for convergence guarantees. Consequently, the unwanted iterations heavily degrade work efficiency. On the other hand, an *ordered* execution follows strict priority constraints, where the highest priority tasks are always processed ahead of low priority tasks. This execution mode improves work efficiency. However, previous work (such as KDG [12]) has shown that synchronization and communication efforts required for ordered execution outweigh the performance benefits. To overcome this, Swarm [14] proposes speculation across ordering constraints using specialized hardware. Swarm achieves super-linear speedups over sequential implementation at the cost of high hardware overheads.

Previous works [10], [9] have shown that the task execution can follow a partial priority order to improve performance instead of ignoring priority order. In this paradigm, a concurrent priority scheduler (CPS) data structure selects high-priority tasks and schedules them to different cores for parallel processing. CPS reduces the communication burden by selecting high-priority tasks instead of the highest priority task at an execution instance. Following a relaxed priority order, a CPS exploits multicore parallelism for improved performance. However, due to the potential divergence of priority ordered task processing across cores, a CPS may end up executing redundant tasks. Therefore, a good CPS design aims to select high-priority tasks with low communication among the cores to improve work efficiency of the underlying algorithm.

A CPS can schedule tasks among the cores in two ways. In a pull style CPS, a core pulls tasks from a global worklist, or steals/requests work from other cores when it is out of work. This mode minimizes communication since a core only pulls work from other cores on demand. However, minimizing communication can lead to divergence in task priorities being processed by the cores. In a push style CPS, each core continuously distributes tasks to other cores to ensure high-priority task propagation and load-balanced execution. However, this mode of task distribution leads to high communication cost among the cores.

A. State-of-the-art CPS designs

A recent study conducted an empirical performance analysis of modern CPS designs [10]. It concluded that Galois, ordered by integer metric (OBIM) scheduler, delivers high performance followed by RELD. OBIM is a pull-style,

relax-ordered distributed priority scheduler that implements a coarse-grain task distribution model. It merges tasks with priorities in close range into a single priority distributed and unordered bag. OBIM stores the bag metadata in a shared global map data structure. Whenever a core runs out of work, it first searches for the highest priority bag in the global map and then process all tasks in that bag. Moreover, each core adds newly created bags to the global structure. Following this approach, OBIM schedules a few bags instead of many tasks, which leads to reduced communication cost.

In OBIM, processing bags at fixed-size granularity leads to tasks with diverging priorities in a bag, which degrades work efficiency. PMOD [10] addresses the fixed bag size constraint of OBIM. It dynamically estimates the usage of bags at runtime. It merges and divides the bags based on application behavior to prevent under- and over-utilization of bags. Both OBIM and PMOD serialize task transfers and processing that leads to degraded performance. Minnow [13] introduces decoupling task transfers from processing using hardware support. It introduces a per-core helper minnow core to offload task (work-list) scheduling and pre-fetching to improve communication cost. However, the performance benefits come at the high hardware cost of dedicated minnow cores.

RELD [10] is a push-style CPS that implements a fine-grain task distribution model. It maintains a distributed array of concurrent priority queues, where each priority queue (PQ) is associated with a core. Each PQ is a software data structure that stores tasks, and maintains ordering based on task priority. The dequeue operation returns the highest priority task available in the PQ. Each core dequeues a task from its PQ, executes it, and distributes the generated children tasks to other cores by selecting a remote core at random. The continuous task distribution aims for load-balanced execution, while keeping cores from diverging on their execution of high-priority tasks. However, this approach creates significant communication overhead.

All aforementioned CPS designs search high-priority tasks, but they are unaware that cores may drift in priority. Without explicit tracking of task priorities, a CPS cannot adapt and compensate for the drift. Moreover, even if the cores are not drifting, the CPS is unaware of the communication cost required for efficient execution. A CPS design should track priorities of tasks and use it as a signal to co-optimize drift and communication at runtime [15]. Adapting for priority drift improves work efficiency of the algorithm, which improves both communication and computation costs. Moreover, if priority drift does not improve work efficiency, the drift signal is helpful to optimize communication by preventing unnecessary task transfers.

B. A case for priority drift aware CPS design

We quantify the priority drift of a core as the average absolute difference of priority between its highest priority task (P_0), and the highest priority of tasks being processed by all the cores at a given time instance. A formal definition of priority drift is shown in Equation 1 for N cores.

$$Priority\_Drift = \frac{\sum_{i=1}^{N} abs(P_0 - P_i)}{N} \quad (1)$$

The use of priority drift seamlessly integrates in a push style CPS design (RELD) since it implements fine-grain controls for the rate of task distribution among the cores. On the other hand, use of priority drift in a pull-style CPS design (like OBIM and Minnow) is limited since it does not offer direct control over adjusting the rate of task distribution among cores. Therefore, we choose RELD as a starting point for HD-CPS to optimize priority drift among cores using hardware-software co-design for work-efficient execution. It implements the following software methods.

- A novel feedback-driven runtime heuristic that computes priority drift among cores at different coarse-grain time intervals. The heuristic allows each core to independently adapt task distribution to minimize priority drift among cores, improving work efficiency and communication cost.
- Decouple task transfers from task processing using a dedicated per-core software receive queue, enabling fast propagation of tasks and improving priority drift.
- Dynamically cluster tasks with similar priorities into bags that reduce task processing overheads, which improves priority drift.

The proposed HD-CPS software capabilities show that PQ operations and task transfer costs are the main contributors to the execution time. The following hardware methods are proposed to mitigate these overheads.

- Implement a per-core hardware receive queue to accelerate task transfers. Further, the task transfers rely on hardware messaging support in the on-chip network to achieve low task transfer latency. Accelerating task transfer helps lower priority drift among cores. Prior works have implemented hardware messages for task transfers (e.g., [16], [17], [18], [14]), including commercial multicore processors [19], [20].
- Implement a per-core hardware priority queue to accelerate PQ operations. The low latency enqueue and dequeue operations enable fast processing of tasks, which improves priority drift among cores.

These proposed HD-CPS software and hardware methods are described next.

III. HD-CPS Architecture

A. Decoupling task transfer from task processing

HD-CPS uses software distributed per-core priority queues (PQ) to store tasks. Each core continuously distributes tasks among the cores, which leads to load-balanced execution and fast propagation tasks. However, the PQ is

used for both task transfer (enqueue) and processing (dequeue). Since a core can perform remote enqueue operation on any core's PQ, it must be done atomically. Consequently, a core must lock its priority queue to perform the dequeue operation. Both enqueue and dequeue operations are time-consuming, resulting in a rebalancing of the priority queue. The blocking nature of these atomic PQ operations obstructs the cores from processing and distributing tasks. These time-intensive atomic operations prevent the propagation of high-priority tasks among the cores and reduce the task processing rate.

HD-CPS decouples task transfers from task processing. Instead of using a single PQ per core for both these operations, it uses two software queues per core, a receive queue for incoming tasks and a priority queue for selecting high-priority tasks for processing. The *receive queue* for incoming tasks is a circular list and contains a finite number of entries. Each slot comprises a flag and a placeholder for task metadata. A sending core atomically increments the corresponding receive queue's write pointer in the destination core, then places its data into the slot and sets the flag. The atomic increment on the pointer makes sure that multiple cores do not write to the same slot. Each core processes new task(s) in its receive queue with high priority, and moves them to its priority queue for future processing. Unlike the atomic operations on the priority queue, the receive queue's atomic operations are faster while the priority queue is relieved from processing enqueue and dequeue operations atomically. Figure 1:① shows the proposed decoupled per-core receive queue and its task transfer flow, as well as its interface with a core's priority queue.

This separation of task transfer and processing allows the sender core to transfer tasks faster. Moreover, a receiver core eliminates the blocking atomic operations on its priority queue, allowing for faster processing of high-priority tasks. In turn, priority drift improves among the cores, resulting in improved performance.

B. Adaptive processing of tasks and bags

Performing PQ operations at per task granularity can result in high PQ overheads. A PQ returns the highest priority element on a dequeue operation. The addition and removal of elements from the PQ result in re-ordering operations within the queue to ensure this constraint. These ordering operations are compute-intensive and dominate the overall execution time. Moreover, these operations also obstruct the cores from progressing forward, leading to increased priority drift. One way of reducing these PQ overheads is to decrease the number of PQ operations. HD-CPS uses bags of tasks to overcome these overheads. Tasks with the same priority are bundled together in bags (similar to the concept of bags in OBIM). The bag consists of the payload for its tasks and metadata to track the bag identifier and its priority. Only the bag metadata is enqueued in the core's PQ, while the

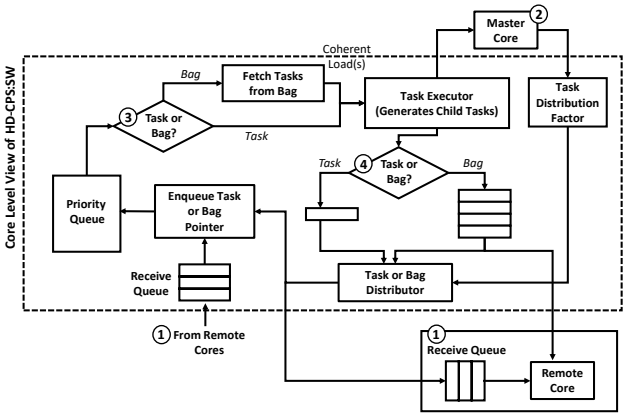

Figure 1: Core Level view of the HD-CPS Architecture.

payload is either stored at the sender or the destination core's side. The PQ at the destination core dequeues the bag metadata when it becomes the highest priority at that core. The storage of bag payload offers two implementation options. (1) Store the bag payload at the sender core and use coherent loads to retrieve data when the bag is dequeued from the PQ. (2) Communicate bag payload alongside its metadata and store payload and metadata at the destination core's PQ. Both these options are evaluated in the evaluation section. However, the coherent loads option delivers better performance since it transfers a bag's payload on-demand and exploits inherent data locality in large data payload sizes.

Algorithm 1 Heuristic For Creating Bags of Tasks

Algorithm:

```
1:  while (PQ ≠ ∅) do
2:      task = Q.pop ()
3:      children_tasks = PROCESS_TASK(task)
4:      <priorities, task_ids> =
        COUNT_PRIORITY(children_tasks)
5:      for priority in priorities do
6:          if priority.count within threshold then
7:              bag = CREATE_BAG(task_ids)
8:              SEND(bag)
9:          else
10:             SEND(tasks)
```

Selecting bag or individual task for processing: HD-CPS utilizes bags when it is beneficial to process proximity priority tasks together. It implements a heuristic to transfer an individual task or a bag at runtime. The highest priority task (or bag) is executed to generate children tasks on each dequeue of a core's PQ. Algorithm 1 shows the modified dequeue operation in a core. Instead of sending all children tasks one by one to their destination PQs, HD-CPS bundles the tasks with approximate priorities into bags (Algorithm 1, line 7–8). However, children tasks that do not fall in a

priority range are still distributed at the task granularity (Algorithm 1, line 9–10). HD-CPS chooses a bag when the payload size exceeds a threshold ($\geq$3 but $<$10 tasks used in this paper), while it distributes the tasks individually otherwise (Algorithm 1, line 6). The latter is chosen when the payload is small; thus, it is beneficial to distribute tasks individually and insert them in their destination PQ. HD-CPS also puts an upper bound on the bag size. If the bag size is quite large, it can bind the core to process all assigned tasks even if high priority tasks/bags are available in cores' PQ. Therefore, HD-CPS uses an empirically determined upper bound to make the system robust against this issue. This adaptive behavior avoids unnecessary computations associated with handling individual tasks as bags, i.e., insert bag metadata in a PQ, and then separately retrieve its payload on dequeue. Figure 1:③ and ④ visualizes the flow of task and bag at the core level in HD-CPS.

C. Priority drift-aware task distribution among cores

As discussed in Section II, work-efficiency and communication are both directly related to priority-drift. A high priority drift indicates that not all cores process high-priority tasks, which degrades work efficiency. Thus, if the priority drift is deteriorating during execution, the rate of task transfers among the cores must be adjusted to improve it. However, this value needs to be a non-zero number as the task distribution improves load balance among cores, even when it incurs communication cost. Moreover, it is also essential to be aware that increasing the task distribution rate may not help improve priority drift at some point. Therefore, aggressive task distribution without motioning priority drift can lead to superfluous communication overheads. Moreover, it does not account for the fact that increasing communication also obstructs the cores from processing tasks, ultimately increasing priority drift.

Feedback drift-aware heuristic for task distribution: HD-CPS proposes a history-based heuristic for task distribution that optimizes both priority drift and communication cost. The objective is to keep the priority drift low while preventing unnecessary traffic into the on-chip network. The heuristic quantifies task distribution using a metric *task distribution factor (TDF)*, which is defined as the ratio of remote enqueue operations and the total number of enqueues performed by a core. For instance, consider a TDF of 75%, then the core will send three tasks out of every four enqueued tasks to random cores, and the remaining one task will be inserted into its own priority queue.

The heuristic measures priority drift among the cores at different coarse grain time intervals and compares it with the previous interval's priority drift to determine the next interval's task distribution factor. It also keeps track of whether TDF was increased or decreased in the previous sampling period. To calculate priority drift, after processing a fixed number of tasks (2K in this paper), each core sends the priority of its latest task processed to a dedicated core (c.f. Figure 1:② for core level view, and Algorithm 3 for the decision flow). After receiving task priorities from all cores, the dedicated core calculates the relative priority differences to get the average priority drift (c.f. Algorithm 2, lines 1–3.). It then compares the calculated priority drift to the previous interval's priority drift to decide about its TDF value. A positive difference implies priority drift is getting worse and leads to two scenarios. If the TDF was increased in the previous interval and the latest drift worsened, then the next TDF is decreased as increasing communication did not help the priority drift (c.f. Algorithm 2, line 5–7). Similarly, if the TDF was decreased in the previous interval and the latest drift worsened, the next TDF is increased to improve the priority drift (c.f. Algorithm 2, line 8–10). A negative difference implies priority drift is getting worse. However, in this case, the TDF is always increased as the heuristic's goal is to optimize priority drift (c.f. Algorithm 2, line 11–13).

A key advantage of this heuristic is that it is non-blocking. It does not block the remote cores as they proceed with an

Algorithm 2 Task Distribution Factor calculation

TDF: Task Distribution Factor
pd: Average priority drift of current interval
pd_prev: Average priority drift of previous interval
$decision\_prev$: Previous decision to increase or decrease tdf
Algorithm:

```
1:  for all (cores) do
2:      pd += ABS(remote_priority - master_priority)
3:  pd = pd / NUM_CORES
4:
5:  if (pd ≥ pd_prev AND decision_prev == increase) then
6:      TDF = TDF - 1
7:      decision_prev = decrease
8:  else if (pd ≥ pd_prev AND decision_prev == decrease)
    then
9:      TDF = TDF + 1
10:     decision_prev = increase
11: else if (pd ≤ pd_prev) then
12:     TDF = TDF - 1
13:     decision_prev = decrease
14:
15: pd_prev = pd
```

Algorithm 3 Transfer of latest priority value to master core

$send\_threshold$: Sampling interval.
$processed$: Number of tasks processed
$master\_id$: Master core id
PQ: Priority Queue
Algorithm:

```
1: while (PQ ≠ ∅) do
2:     task = Q.pop ()
3:     PROCESS_TASK(task)
4:     processed = processed + 1
5:     if (processed == send_threshold) then
6:         SEND(master_id, task.priority)
```

old value of TDF until the new value propagates to them. However, it does require computations on the dedicated core's side for TDF decisions. Thus, the granularity at which HD-CPS updates TDF needs to be selected empirically, as discussed in the evaluation section. Moreover, the initial value of TDF also affects the subsequent decisions of the heuristic. We use an initial value of 50% based on empirical results discussed in the evaluation section.

Adaptive TDF Oracle: To evaluate the efficacy of the proposed adaptive TDF heuristic, an *oracle TDF* is introduced. It starts by sweeping all TDF values for the first sampling period, and selects the best TDF that results in the highest performance. The algorithm is executed again with this optimal TDF for the first interval. But the TDF is now swept again for the second interval to select its optimal TDF. Continuing in this manner, the dynamic oracle finds the best TDF for each interval iteratively, and creates a history of best TDF values for each sampling interval. The algorithm is then executed with these best TDF values at each sampling interval and the completion time is measured. The oracle only serves as a method to compare performance against the proposed adaptive TDF heuristic.

D. Hardware acceleration of task transfer & processing

As discussed in Section III-A, HD-CPS uses a per-core receive queue to decouple task transfers from task processing. However, this software approach keeps the cores' pipeline busy as it requires computation to transfer the task into a remote core's receive queue. Moreover, the software-based priority queue operations (enqueue and dequeue) significantly contribute to the total completion time of a task-parallel program. The reason is that these operations block the cores from processing and distributing tasks, where cores are occupied in ordering the priority queue elements for fast priority access.

HD-CPS mitigates the challenges mentioned above using software-backed hardware queues. It implements a dedicated per-core hardware receive queue, hRQ. Moreover, it also employs hardware-based non-blocking core-to-core messages to transfer tasks between cores [19]. The hRQ along with fast hardware messages lowers communication cost and improves task propagation and transfers, thus optimizing priority drift. HD-CPS also implements per-core hardware priority queue, hPQ, which offers much lower latency than its software counterpart and accelerates enqueue/dequeue operations. The number of entries in both these queues is limited as these queues act as buffers for already existing software queues. The interactions between software and hardware queues is core local, and does not require any global intervention.

Lifetime of a Task: A core receives a task when it performs an local enqueue operation (c.f. Figure 2: ⓵ⓐ), or a remote enqueue from another core (c.f. Figure 2: ⓵ⓑ , ②). Remote enqueues are done using hardware messages in

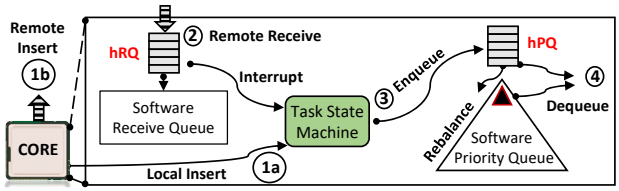

Figure 2: hRQ and hPQ hardware queues in HD-CPS.

the on-chip network [19]. These messages are asynchronous; thus, the sender core can proceed forward after injecting a message into the network (c.f. Figure 2 ⓵ⓑ). On the receiver core side, every incoming message is added to the hardware receive queue (c.f. Figure 2②). This step does not require any software intervention from the core. If the hRQ is full, the incoming task(s) are pushed in the software receive queue. An interrupt is flagged whenever a remote message arrives at a core, triggering an interrupt service routine (ISR). The core invokes the *task state machine* to transfer tasks from the hardware-software receive queue to the priority queue using the ISR. This interrupt is serviced with high priority when the core is not executing its PQ-related operation.

The enqueue of tasks in the hPQ is handled by the task state machine (c.f. Figure 2:③). The goal is to efficiently utilize the hardware entries in the hPQ since the number of entries is finite. If the hPQ is full, the least priority task is evicted from the hPQ to make space for the incoming task. Consequently, the evicted task is moved to the software priority queue. The core continues its future operations without waiting for the software task queue to rebalance its elements, as it is done by dedicated logic. Thus, the rebalancing operation is asynchronous with the core's pipeline. In essence, the hardware priority queue acts as a buffer to hide the rebalance latency. In the case of low hPQ utilization, all tasks fit and result in fast enqueue latency. However, when priority queue utilization is high, the buffering nature of hPQ advocates hiding the latency overheads of balancing the software priority queue.

On a dequeue operation (c.f. Figure 2:④), the core removes the highest priority task from the hardware or software priority queue. If tasks are present in the software queue, the dequeue operator checks the balanced software queue with a constant latency. This is done in parallel to the dequeue operation on the hardware queue. The highest priority task from the software queue is compared with the one acquired from the hPQ, after which the highest priority one is selected to be returned to the core pipeline. If the task is removed from the software queue, it needs to be rebalanced. Here again, the *task state machine* makes sure there is no pending balancing operation. If there is a pending operation, the core stalls; otherwise, the core continues its work while the software queue gets rebalanced

in the background.

Queue sizing and utilization: The size of both hardware queues has an impact on performance. However, it is difficult to predict the queue sizes as their usage depends on the algorithm and input. For example, the usage of hPQ is relatively high for dense and large graphs. As these factors are difficult to control, the size of both these queues is determined empirically, optimizing both performance and hardware overheads. Our evaluation has chosen the sizes 32 and 48 for hRQ and hPQ respectively.

The size of each entry in the hRQ and hPQ is 128 bits. Each entry contains two fields, ID and data, each of which is 64 bits. The total hardware overhead for a 32 entry hRQ and 48 entry hPQ is $1.25KB$ per-core. If the size of both these queues is set to zero, then the system becomes a software-only solution without any hardware acceleration.

Flow control of hardware messages: The hardware receive queue is backed up in software; therefore, there is no need for precise flow control. However, HD-CPS uses a flow control mechanism to prevent over-utilization of hRQ. Each core maintains an array of shared hardware flags, *capacity counter* for all other cores. When a sender core chooses a random destination core, it checks the corresponding capacity counter atomically. If the flag is set, the sender core chooses some other core to send the message. However, if the flag is not set, the core sets the flag and sends the message. The destination core is responsible for moving the tasks in its receive queue to the priority queue. Whenever the task is moved to the priority queue, the sender core's capacity counter flag is cleared to send other tasks.

Termination Condition: Whenever both priority queue and receive queue of a core are empty, it broadcasts its status to all other cores in the system. Moreover, the core also checks the status of these queues for all other cores. Each core terminates when all other cores are out of work. As the hardware messages are asynchronous with the core's pipeline, a message can be in transit when the termination check happens. HD-CPS addresses this situation by using the hardware messages flow control mechanism. Each core also checks its capacity counters to ensure no incoming message are outstanding.

IV. Methodology

A 40-core **Intel Xeon E5-2650 v3** multicore CPU with 4 sockets, and 10-cores per socket is used for evaluation. The machine has $512GB$ DDR4 RAM and a $25MB$ L3 last-level cache. All benchmarks use the `pthreads` library to utilize up to 80 threads and are compiled using the g++ compiler (v 6.4.1).

The hardware queues and messages in HD-CPS, as well as Minnow and Swarm are implemented using an in-house industry-class **RISC-V multicore simulator** [21]. A 64-core tiled multicore processor with a two-level coherent private L1, shared L2 cache hierarchy per core, and a 2D mesh on-chip network with X-Y routing is evaluated. The default architectural parameters used for evaluation are shown in Table I. Task management instructions are added to the ISA, which include *enqueue* to add a task to the task queue using core-to-core hardware messaging support. The *dequeue* instruction pops a task with the highest priority from the priority queue. The results from the simulator are also correlated with **Tilera®Tile-Gx72™** multicore processor [19].

Number of Cores	64 RISC-V, In-Order @ 1 GHz
Memory Subsystem	
L1–I, LD–D Cache per core	32 KB, 4–way Assoc., 1 cycle
L2 Inclusive Cache per core	256 KB, 8–way Assoc.
Directory Protocol	Invalid–based MESI, ACKwise$_4$
DRAM Controllers	8, 10 GBps per Contr./ 100ns
Electrical 2–D Mesh with XY Routing	
Hop Latency	2 cycles (1–router, 1–link)
Contention Model	Only link contention, 64 bit Flits (Infinite input buffers)
Hardware Queues	
Per-core Queue Entries	32 hRQ, 48 hPQ entries
HW Queue Latency	5 cycles per access
Task and Bag ID Size	128-bits

Table I: Multicore simulator parameters for evaluation.

A. Software CPS Designs

All software CPS designs are evaluated using the 40-core Intel Xeon machine. Open-source implementations of **OBIM and PMOD** [9], [10] are used from the Galois framework (version 2.2.1). **Software Minnow** [13] is implemented on top of OBIM by partitioning the total number of available cores in the processor into two groups. The first group executes the worker threads that perform task processing operations. The second group executes Minnow threads responsible for prefetching task data, and storing generated tasks (bag) into data structures used by the associated worker threads. Software Minnow is implemented using 36 worker and 4 minnow cores in the Intel Xeon machine. This selection process is empirically evaluated in Section V-C.

The open-source implementation of **RELD** is used from the Galois framework release of PMOD [10]. **HD-CPS:SW** uses the RELD implementation as its starting point, but it is implemented in the latest version 5 of the Galois framework. The following software configurations are evaluated for HD-CPS.

- **sRQ** implements decoupling of task transfer from processing using the method outlined in Section III-A.
- **sRQ + TDF** adds priority drift heuristic on top of sRQ as discussed in Section III-C.
- **sRQ + TDF + AC** always uses bags on top of sRQ + TDF as mentioned in Section III-B.
- **sRQ + TDF + SC** uses the heuristic from Section III-B to select tasks or bags. This configuration is also

referred to as **HD-CPS:SW**.

B. Hardware assisted CPS Designs

In **Minnow** [13], each worker core is paired with a dedicated minnow helper core that performs the bag prefetch operations. These hardware capabilities are modeled in the RISC-V multicore simulator. **HD-CPS:HW** is also modeled in the simulator by adding the following hardware capabilities on top of HD-CPS:SW.

- **hRQ** implements hardware receive queue for task transfers as mentioned in Section III-D.
- **hRQ + hPQ** implements hardware priority queue on top of hRQ as discussed in Section III-D. This configuration is also referred to as **HD-CPS:HW**.

Swarm [14] exploits task-parallelism for strictly ordered algorithms that go well beyond a CPS design's scope. It employs speculative task execution in hardware, where tasks are processed out-of-order but always committed in-order. At commit time, if a task is determined to have violated task ordering constraint during its execution, the task along with all children tasks originating from this task are recursively killed. This process requires reverting all memory modifications made by the task being killed, as well as its children tasks. Although this approach results in super-linear speedup compared to sequential implementation, it requires multiple per-core task, order, and commit queues. This results in 10s of kilobytes of hardware overhead per core, as well as complex parallel lookup logic. Swarm is also modeled in the RISC-V multicore simulator.

It is non-trivial to outperform Swarm due to its highly sophisticated hardware for out-of-order task execution. However, HD-CPS:HW is compared against Swarm to demonstrate that it can reach Swarm's level of performance using significantly less hardware.

C. Breakdown of Completion Time

The performance of each CPS design is measured by tracking the parallel completion time. To gain insights, the completion time measurements are also tracked as follows: **enqueue** incorporates the time to enqueue a bag or a task, and time spent in creating bags; **dequeue** is the time to dequeue a task or a bag (and tasks inside it); **compute** is the time to process tasks by a core; **comm** is the time spent in transferring tasks and the time spent while the core is idle. For Swarm, the cost of rollback is shown as part of the compute component.

D. Evaluation Benchmarks

The following task-parallel graph benchmarks from the PMOD [10] baseline are used for evaluation. Each benchmark picks its best performing sequential baseline and state-of-the-art parallel, shared memory implementation.

Single Source Shortest Path (SSSP) algorithm uses Delta-Stepping [22] to find the minimum distance paths from a source vertex to all vertices in a weighted graph. Each task processes a vertex whose priority is its distance from the source node, with lower distances having a higher priority.

A* Shortest Paths (A*) is a search algorithm that utilizes a heuristic to guide its search [23]. Like SSSP, each task processes a vertex, and its priority is the sum of its distance from the source node and heuristic's value.

Breadth-First Search (BFS) starts from a source vertex and searches vertices in a graph using the edge first method with the weight of each edge set to one [24]. Each task processes a vertex whose priority is its distance from the source node, with lower distances having a higher priority.

Minimum Spanning Tree (MST) uses Boruvka's algorithm [25] to find a spanning tree over all vertices with minimum total edge weight. Each task processes a vertex, picks the minimum weight edges, and merges them with the corresponding neighbor. Each merge results in a new task and is prioritized by its degree.

Graph Coloring (Color) implements vertex coloring based on their saturation degree [26]. Each task processes a vertex and tasks are prioritized by their degree.

PageRank determines the rank of a vertex in a graph. This work uses *push-pull* version [27] that calculates a vertex's rank by evaluating incoming edges and propagating the change to the vertices associated with outgoing edges. Tasks are prioritized according to their rank using integer numbers to make them compatible with OBIM.

Inputs	Nodes	Edges	Avg. Deg.	Max. Deg.
CAGE14 [28]	1.505M	234M	34	80
rUSA [29]	24M	58M	1.2	9
Web-Google [30]	875k	5M	11	6.4K
LiveJournal [31]	4.8M	69M	28	20k

Table II: Input graphs and their respective statistics.

E. Evaluation Inputs

Table II shows the evaluated directed graph inputs and their characteristics. These graphs represent varying degrees, densities, and sizes. We note that the size of hardware queues does not depend on graph size; instead, it depends primarily on the density of the graph. We evaluated several dense graphs of large size (e.g., Twitter graph), and found our results to be consistent with the evaluated inputs, CAGE14, LJ, and WG. For a dense graph, a parent task generates more children tasks than a sparse graph. However, as long as the receive queues are cleared at a reasonable rate, the core can accept new tasks and keep the priority drift low. Dense graphs also take advantage of bags, thereby reducing communication cost and keeping the receive queue utilization low. In addition to dense graphs, we also evaluate a sparse representative graph from the USA road networks.

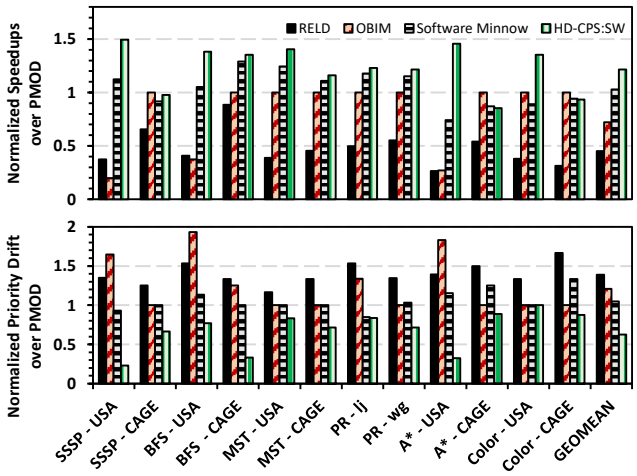

Figure 3: Completion times and Priority drift normalized to PMOD on Intel Xeon.

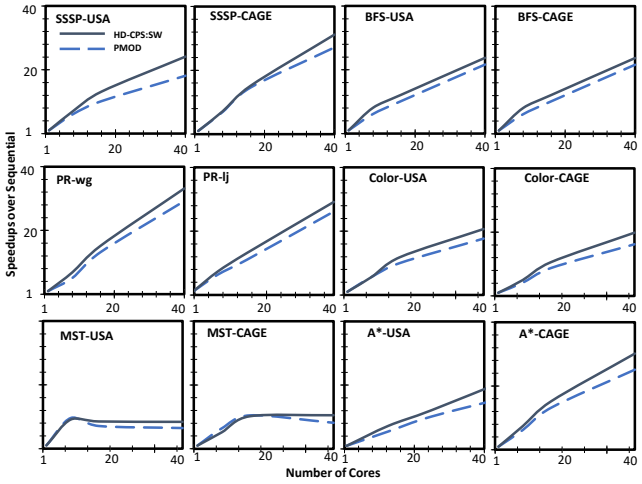

Figure 4: Performance normalized to optimized sequential implementation on Intel Xeon. X-axis shows the number of threads.

V. Evaluation

Figures 3 shows the performance evaluation of RELD, OBIM, Software Minnow, and HD-CPS:SW normalized to PMOD on the Intel Xeon machine. The figure also shows the average priority drift computed at different fixed sampling intervals during the execution of each workload. RELD is not aware of priority drift, which leads to unnecessary, redundant task executions. Second, it suffers from unwanted communication due to constant distribution of tasks among the cores. The results show an average of 1.4× increase in priority drift that translates to more than 2.2× performance loss compared to PMOD. In OBIM, when bags are under-utilized (e.g., SSSP, BFS, and A* with USA graph), the priority drifts among cores, resulting in performance loss compared to PMOD. PMOD optimizes priority drift over OBIM using better bag utilization. Software Minnow exploits parallelism in OBIM by hiding the latency of task transfers. However, it also suffers as some cores are allocated as minnow cores. The loss of computing power in Software Minnow hinders performance. Overall, both PMOD and Software Minnow improve priority drift, and thus performance over OBIM.

HD-CPS:SW tracks and adapts to improve priority drift across cores. In general, the improvement in priority drift translates to consistent performance improvements. The gains in performance may not correlate with gains in priority drift across different workloads as the priority's role is different for each workload. Moreover, even if the priority drift does not improve, HD-CPS:SW manages communication cost better by dynamically adapting the task distribution rate (e.g., Color-USA). Overall, HD-CPS:SW improves performance over PMOD and Software Minnow by 1.25× and 1.12× respectively.

Figure 4 shows the performance scaling with increasing core counts for PMOD and HD-CPS:SW on the Intel machine. HD-CPS-SW consistently performs better or at par with PMOD, and its performance increases with increasing core counts. This behavior is primarily attributed to higher communication cost at higher core counts that HD-CPS-SW is able to overcome better with its explicit tracking of priority drift among cores.

Figure 5 shows the completion time breakdowns and priority drift of different workloads with HD-CPS:SW variants normalized to RELD. sRQ accelerates task transfers using a per-core software receive queue and leads to two cases. (1) It improves priority drift and subsequently improves work efficiency, which reduces the number of tasks processed. Therefore, sRQ improves all the breakdown components (e.g., SSSP-USA, A* USA, BFS and, PR). (2) It improves priority drift, but the amount of work done does not change significantly. However, this case shows improved communication delays (e.g., SSSP-CAGE, MST, Color, and A*-CAGE). sRQ improves performance over RELD by 1.3×.

$sRQ+TDF$ uses priority drift to balance communication and the total amount of work. In some cases, it improves the priority drift significantly and thus improves work efficiency (e.g., SSSP-USA, A* USA. BFS and PR). However, adapting TDF prevents aggressive task distribution in other cases, thus improving communication cost (e.g., SSP-CAGE, A*-CAGE, MST, Color). $sRQ + TDF$ improves performance over RELD by 2×.

$sRQ+TDF+AC$ always creates bags and is beneficial when each parent task creates several children tasks (e.g., SSSP-CAGE). However, it hurts performance when a parent task creates few children tasks (e.g., SSP-USA, A*-USA, MST, and Color). These cases show increased enqueue and dequeue costs due to the additional bag creation overheads.

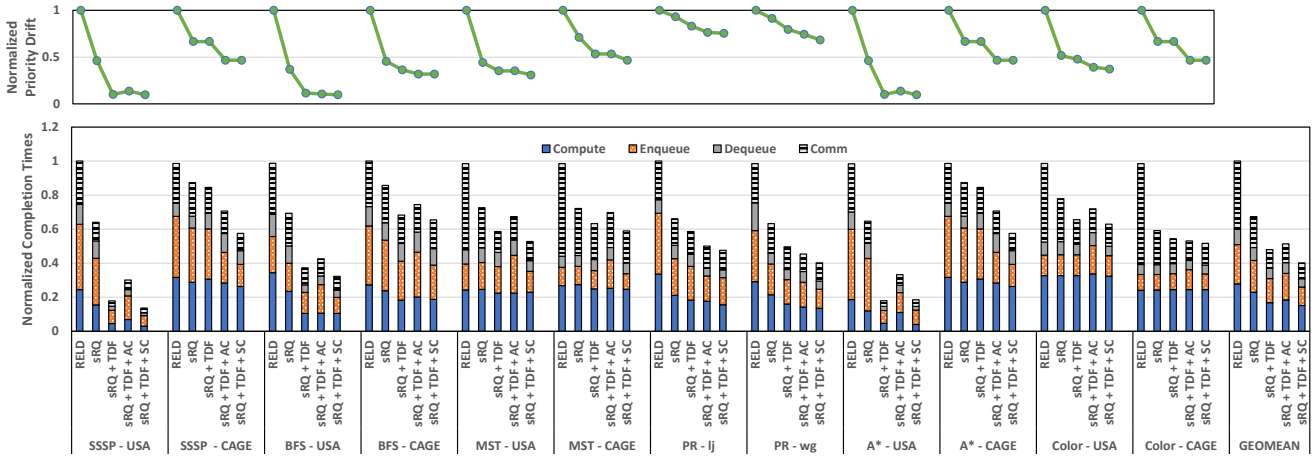

Figure 5: Completion times and Priority drift of HD-CPS:SW variants normalized to RELD on Intel Xeon.

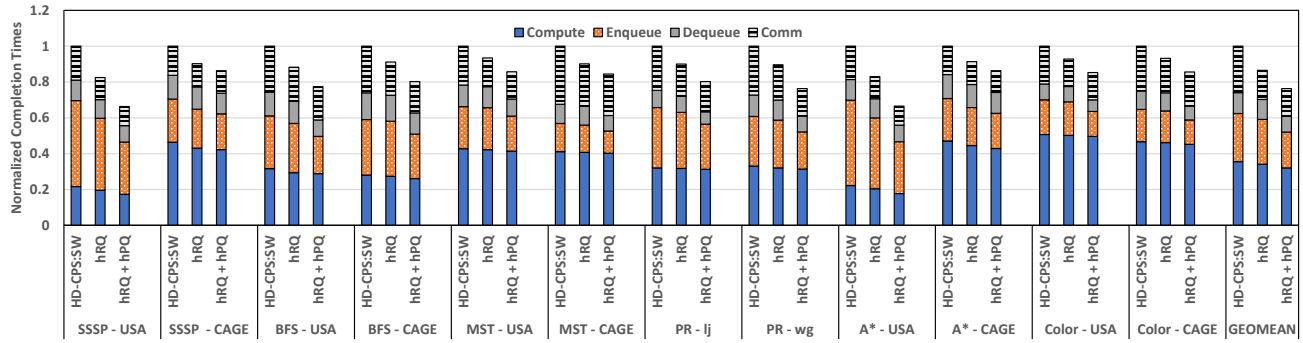

Figure 6: Completion times of HD-CPS:HW variants normalized to HW-CPS:SW on the Simulator.

$sRQ + TDF + AC$ improves performance over RELD by 1.9×, which is worse than $sRQ + TDF$. However, $sRQ + TDF + SC$ selectively creates bags and overcomes the shortcoming of $sRQ + TDF + AC$. It significantly reduces enqueue and dequeue overheads where bags are helpful (e.g. , SSSP-CAGE, A*-CAGE). $sRQ + TDF + SC$ improves performance over RELD by 2.4×.

A. Evaluation of HD-CPS:HW

Figure 6 shows the completion time breakdowns of different workloads with HD-CPS:HW variants. These breakdowns are normalized to HD-CPS:SW. hRQ lowers communication cost for all workloads due to faster task propagation. The overall improvement achieved by hRQ over HD-CPS:SW is around 10%. The benefits of $hRQ + hPQ$ primarily depend on the priority queue (PQ) utilization. In sparse graphs (e.g., USA), the PQ utilization is around 50 tasks per PQ at any given time. However, for dense graphs (e.g., CAGE), PQ utilization can reach several thousand tasks. Therefore, when PQ utilization is low, the benefits shown by $hRQ + hPQ$ are high. When PQ utilization is high, hPQ still shows good performance benefits due to the buffering and latency hiding nature of hPQ. Moreover, hPQ allows faster transfer of tasks from receive queue to hPQ, allowing the priority drift to stay in check. Overall, $hPQ + hRQ$ improves performance over HD-CPS:SW by 20%.

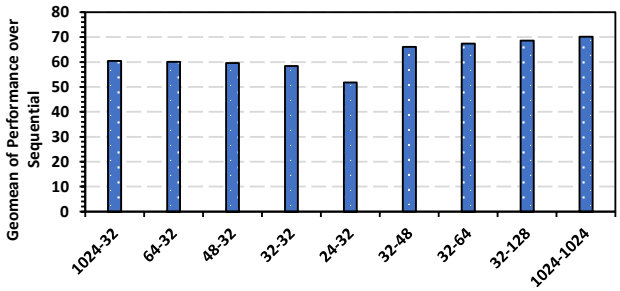

Figure 7: HD-CPS:HW with different queue sizes. The tuple on x-axis shows hRQ size, followed by hPQ size for each configuration.

1) Sizing of Hardware Queues for HD-CPS:HW: Figure 7 shows the geometric mean performance of the evaluated benchmarks using different queue sizes for HD-CPS:HW. The initial five setups fix hPQ size to 32 and decrease

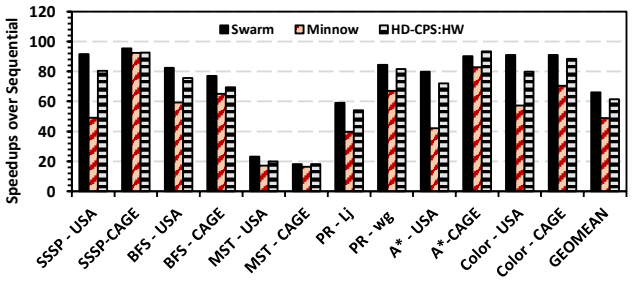

Figure 8: Performance speedup over sequential baseline for Swarm, Minnow and HD-CPS:HW on the Simulator.

the hRQ size from 1024 to 32 entries. We observe that hRQ utilization is approximately 30 or average; hence performance drops when its size decreases from 32 to 24. After tuning hRQ size to 32, the hPQ is increased in the following three setups. Performance improves when hPQ is increased from 32 to 48 but saturates at larger hPQ sizes. This behavior is in line with the earlier observation that PQ utilization is either 50 or in thousands based on the type of the input graph. Based on this empirical analysis, we pick the default size of 32 for hRQ and 48 for hPQ. Since each entry is 128 bits wide, the total per-core overhead of these queues is $1.25KB$.

2) HD-CPS:HW comparisons to Minnow and Swarm: Figures 8 shows the speedups of Minnow, HD-CPS:HW, and Swarm for the evaluated workloads. Swarm maintains strict ordering of tasks during execution, resulting in high work efficiency. It also allows executing tasks out of order, which results in high task-level parallelism. However, it incurs performance penalties whenever speculative tasks are rolled back. Overall, Swarm achieves geometric mean speedup of 66× over sequential workload implementations. Minnow fails to reach Swarm's level of performance as some workloads show low work efficiency due to highly divergent task priorities (e.g., SSSP-USA, BFS-USA, A*-USA). However, for dense graphs Minnow exploits the available task-level parallelism and achieves competitive performance against Swarm. HD-CPS:HW co-optimizes for priority drift and communication cost at runtime, resulting in competitive performance for all workloads against Swarm. Overall, Minnow achieves 48×, while HD-CPS:HW achieves 61× speedup over sequential workload implementations. Swarm edges ahead compared to HD-CPS:HW by ∼7%. However, this performance improvement comes with significantly higher hardware overheads compared to HD-CPS:HW.

Figure 9 shows the completion time breakdowns of Minnow and HD-CPS:HW relative to Swarm. Swarm shows the lowest compute time component compared to other systems as it achieves the highest work efficiency. This behavior is prominent for the USA road network graph, where priority divergence is a challenge. On the other hand, task priorities do not diverge in dense graphs like CAGE. Hence, Swarm is unable to gain performance over the CPS designs. HD-CPS:HW optimizes priority drift as compared to Minnow, which results in higher work efficiency, as evident by the improvements in compute component for most workloads. Minnow improves enqueue and dequeue components due its decoupling of bag pre-fetching from task processing. However, it shows high compute and communication costs due to degraded work efficiency. HD-CPS:HW also optimizes for communication cost at runtime, which allows it to incur lower comm latencies as compared to Minnow. As compared to Swarm, HD-CPS:HW is unable to achieve strict ordered processing of tasks for some workloads, such as SSSP-USA and A*-USA. This results in relatively higher compute, enqueue and dequeue costs. Overall, our evaluation highlights that HD-CPS:HW is competitive against Swarm, and outperforms Minnow by ∼8%.

B. Correlation between Simulator and a Real Machine

The simulator used in the evaluation is modeled after Tilera's 72-core multicore processor. Moreover, the Tilera multicore also supports core-to-core hardware messaging, and per-core hardware receive buffer capability. These features enable the implementation of HD-CPS:SW and hRQ on Tilera. To determine the efficacy of the simulator's performance models, Figure 10 shows the performance evaluation for the simulator versus the Tilera machine. Both HD-CPS:SW and HD-CPS:HW (hRQ only) results show an average performance variation of ∼5%, which validates the efficacy of our simulator results.

C. Analysis of Software Minnow Configurations

Figure 11 shows the performance evaluation of different worker and minnow core configurations on the Intel Xeon machine. For example, the 36–4 configuration implies 36 worker cores and four minnow cores. Each minnow core is responsible for prefetching bags from the work-list for nine dedicated worker cores. For the sparse USA graph, increasing the number of minnow cores improves performance because bag size in this input is underutilized, resulting in many work-list prefetch operations. However, increasing the minnow cores past a certain point shows performance degradation due to the loss of parallelism for worker cores. All other input graphs are relatively dense, resulting in better bag utilization, thus requiring much fewer work-list prefetches. Therefore, Software Minnow benefits from fewer minnow cores and more worker cores in these cases. Overall, we select the 36–4 configuration as it delivers the best geometric mean performance.

D. Analysis of the Adaptive TDF Scheme

Figure 12 shows the performance of HD-CPS:HW and dynamic oracle compared to PMOD. HD-CPS:HW performs at par with dynamic oracle in cases where task priorities are

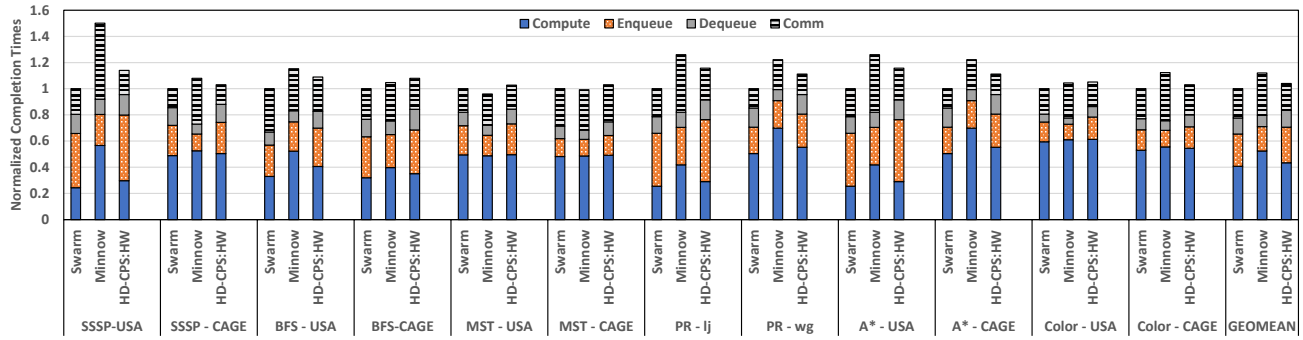

Figure 9: Completion time breakdowns of HD-CPS:HW, Swarm, and Minnow normalized to Swarm on the Simulator.

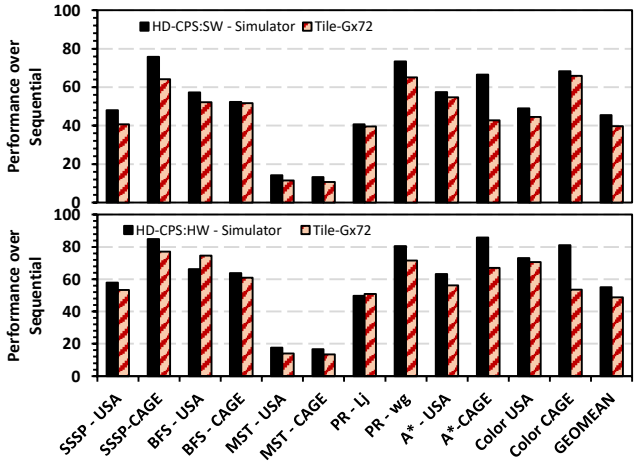

Figure 10: Performance comparisons of Simulator versus Tilera Machine.

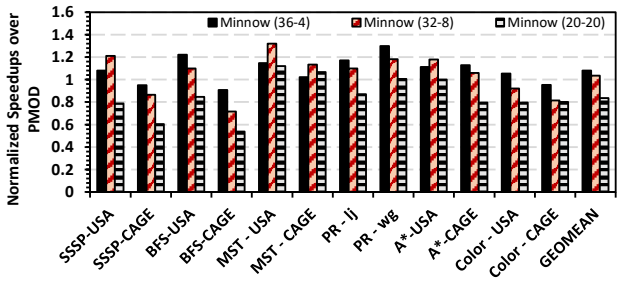

Figure 11: Performance evaluation of different Minnow configurations on Intel Xeon.

close to each other, i.e., most benchmarks that use the CAGE input. However, in cases where priorities are divergent (e.g., SSSP-USA and PR), dynamic oracle shows a slight improvement over HD-CPS:HW. The adaptive heuristic changes the TDF incrementally, while the dynamic oracle swiftly picks the right TDF for a given interval. Overall, the performance of the heuristic is comparable with that of the Dynamic Oracle.

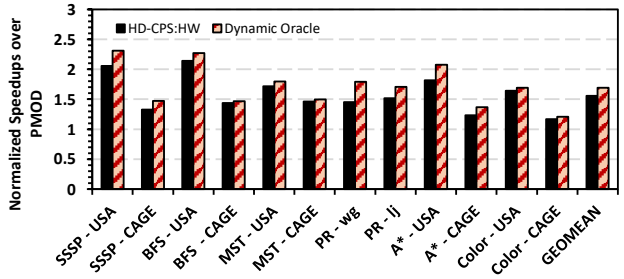

Figure 12: Performance of HD-CPS:HW and Dynamic Oracle normalized to PMOD on the Simulator.

E. Analysis of HD-CPS:SW with different tunable parameters

The priority drift-aware task distribution mechanism described in Section III-C uses a sampling interval parameter that determines when to re-calculate priority drift and update TDF. We use an interval size of 2000 tasks using an empirical evaluation presented in Figure 13:A. For large interval sizes (e.g., 2500 tasks), the priority drift begins to drift before the TDF can be adjusted, leading to performance loss. However, for too small an interval size (e.g., 100 tasks), the computation costs negate the benefits of fine-grain increments. Another tunable parameter is the initial TDF, which determines the TDF for the first interval. As observed in Figure 13:C, the choice of initial TDF does not dramatically affect the final results as HD-CPS quickly corrects it according to the priority drift. We select an initial TDF of 50% since it is easier for the heuristic to move on both sides of the TDF spectrum. The step size determines how much TDF change is desirable to update it. A too low TDF change (e.g., 5%) induces unnecessary oscillations, while a too high TDF change (e.g., 30%) may miss opportunities to optimize priority drift. We use a 10% step size using the empirical study presented in Figure 13:B.

As discussed in Section III-B, HD-CPS can transport the bag's payload data to a remote core in two ways, i.e., over the network (push) or via coherent loads upon

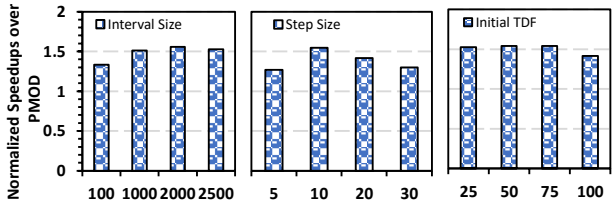

Figure 13: Performance Evaluation of HD-CPS with (A) Sampling Point, (B) Step Size, and (B) Initial TDF of Adaptive TDF Heuristic normalized to PMOD on the Simulator.

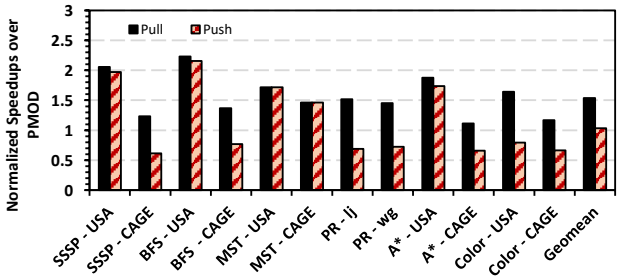

Figure 14: Performance Evaluation of HD-CPS with different bag transport methods normalized to PMOD on Simulator.

priority queue's dequeue of a bag (pull). The push scheme suffers from preemptive creation and transport of each bag's payload data over the network, while the pull scheme is efficient because it only retrieves data when a core needs it to process the bag. The empirical evaluation of both schemes is presented in Figure 14. Although the push scheme performs at par with PMOD, the pull scheme delivers 1.5× better performance. Therefore, it is used in HD-CPS to retrieve the bag's data payload.

HD-CPS also parameterizes selecting the minimum number of generated tasks with the same priority to create a bag. To understand its effect, Figure 15 shows the performance evaluation for a different threshold number of tasks. For a task count of 1, HD-CPS always creates bags. Similarly, for a task count of 5, HD-CPS creates a bag when at least five tasks are generated with the same priority. This parameter is workload-dependent, where the task count threshold depends on the number of tasks being generated with the same priority. We use a threshold of 3 since it delivers the best overall performance.

VI. Conclusion

This paper proposes a novel CPS design that co-optimizes work efficiency and communication among cores using priority drift as a signal. A set of dynamically adaptive task distribution heuristics and fast task transfer mechanisms are proposed to minimize priority drift among cores. Moreover, simple hardware primitives are employed to accelerate task transfers and priority queue operations. HD-CPS is shown to

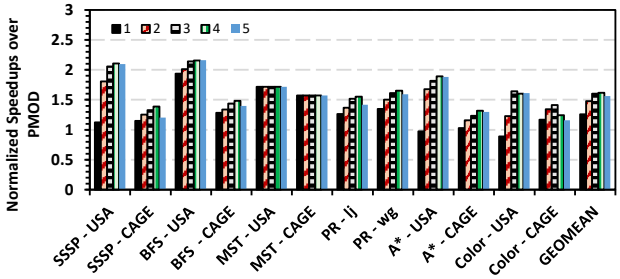

Figure 15: Performance Evaluation of HD-CPS with different bag sizes normalized to PMOD on Simulator.

improve the performance of task-parallel graph algorithms executing on a large core count Intel Xeon machine, and an in-house RISC-V multicore simulator. It outperforms state-of-the-art CPS designs, PMOD and Software Minnow by 25% and 12%, respectively. Moreover, with hardware support, HD-CPS improves performance over Minnow with dedicated minnow cores by 8%, and performs at par against Swarm with significantly lower hardware overhead.

VII. Artifact Appendix

This appendix describes the process of acquiring a subset of experimental results. The artifacts include implementations of various state-of-the-art CPS designs executing parallel graph workloads using real-world inputs. The evaluation is done using an Intel Xeon 40-core CPU with four 10-core sockets, and 512GB of DDR4 memory. However, the artifacts can also be evaluated using other multicore CPU systems, as long as the CPS design parameters are tuned for the underlying hardware platform.

A. Artifact check-list (meta-information)

- **Program:** Concurrent Priority Schedulers: OBIM, PMOD, RELD, Software Minnow, and HD-CPS:SW
- **Algorithm:** SSSP, BFS, MST, and PageRank graph algorithms
- **Compilation:** Galois graph processing framework
- **Data set:** USA road network for SSSP, BFS and MST, and web-google graph for PageRank
- **Run-time environment:** Linux based Operating System
- **Hardware:** Intel shared memory multicore CPU
- **Execution:** Bash scripts for automation of artifacts
- **Metrics:** Completion Time and Priority Drift
- **Output:** Per benchmark completion time and priority drift
- **Experiments:** Subset of Figure 3
- **How much disk space required (approximately)?:** 5GB
- **How much time is needed to prepare workflow (approximately)?:** 6 hours
- **How much time is needed to complete experiments (approximately)?:** 3 hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** BSD 2-Clause

B. Description

1) How to access: The artifact codebase can be downloaded from **https://zenodo.org/record/5794249**

A latest version is also maintained on GitHub https://github.com/RK4/HD-CPS_HPCA-22

2) Hardware dependencies: Intel multicore CPU

3) Software dependencies: The Galois graph processing framework dependencies are listed below.

- https://github.com/IntelligentSoftwareSystems/Galois
- A modern C++ compiler compliant with the C++-17 standard (gcc $\geq$ 7, Intel $\geq$ 19.0.1, clang $\geq$ 7.0)
- CMake ($\geq$ 3.13)
- Boost library ($\geq$ 1.58.0, we recommend building/installing the full library)
- libllvm ($\geq$ 7.0 with RTTI support)
- libfmt ($\geq$ 4.0)

C. Installation

Follow these steps in the prescribed order to complete the artifact installation.

- **Cloning the Artifact Repository** We have created Zenodo and GitHub repositories to clone our artifacts. The repository links are provided in Section VII-B1.
- **Installing Galois** We provide a shell script $compile\_galois.sh$ for fetching and compiling the Galois framework. This script clones Galois release 5.0 from Github, as well as the modified Galois release for the PMOD CPS design. All build files and binaries for Galois are installed in the Galois/build release folder, while PMOD are installed in the PMOD/build folder.
- **Preparing datasets** Run the script $datasets\_fetch.sh$ in the main folder where the artifacts are downloaded. This script fetches the USA road network[1] and web-google[2] graphs, and converts them to the required format for the Galois framework.
- **Installing CPS Designs in Galois** Run the script $install\_cps.sh$ in the main folder. This script adds priority drift related modifications to the OBIM and PMOD CPS designs. Moreover, the Software Minnow CPS design is added using the Obim5.h header file. The RELD and HD-CPS:SW CPS designs are added using the WorklistHelpers_hdcps.h header file. These header files are copied to the appropriate Galois framework's folders.
- **Installing Benchmarks** Run the script $install\_benchmarks.sh$ in the main folder. This script adds each benchmark in the Galois framework and compiles them. Modified benchmarks are provided with the support for all CPS designs.

D. Experiment workflow and expected results

Following the installation steps, each workload is executed by calling the desired {workload name.sh} script. For example, for the SSSP benchmark, execute $sssp.sh$ in the main folder. This script executes all CPS designs for the SSSP benchmark using the USA road network graph, and saves the output in the output/sssp.out file. The benchmarks included in the artifacts are SSSP, BFS, MST, and PageRank.

E. Evaluation and expected results

The Completion Time and Priority Drift metrics are saved in the output folder for each benchmark.

F. Experiment customization

All artifacts are tested using an Intel Xeon machine with 40 cores and $512GB$ DDR4 memory. However, customization parameters for SSSP and BFS are provided for an Intel single socket 8-core machine. Run the script $sssp\_8\_core.sh$ and $bfs\_8\_core.sh$ in the main folder. For further customizations of OBIM and PMOD, the $delta$ parameter must be tuned for the underlying machine. For Software Minnow, the $minCores$ parameter that specifies the number of minnow cores must be adjusted.

[1] http://www.diag.uniroma1.it//challenge9

[2] https://snap.stanford.edu

VIII. Acknowledgment

This work was supported in part by the National Science Foundation under Grant CNS-1718481. This research was also partially supported by the Semiconductor Research Corporation (SRC). The authors wish to acknowledge Brian Kahne from Qualcomm, José A. Joao from Arm, and Masab Ahmad from AMD Research for their valuable feedback.

References

[1] S. Maleki, D. Nguyen, A. Lenharth, M. Garzarán, D. Padua, and K. Pingali, "Dsmr: A parallel algorithm for single-source shortest path problem," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), Association for Computing Machinery, 2016.

[2] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," Technical Report 1999-66, Stanford InfoLab, November 1999. Previous number = SIDL-WP-1999-0120.

[3] A. LUMSDAINE, D. GREGOR, B. HENDRICKSON, and J. BERRY, "Challenges in parallel graph processing," *Parallel Processing Letters*, vol. 17, no. 01, pp. 5–20, 2007.

[4] G. M. Slota, J. W. Berry, S. D. Hammond, S. L. Olivier, C. A. Phillips, and S. Rajamanickam, "Scalable generation of graphs for benchmarking hpc community-detection algorithms," in *ACM International Conference for High Performance Computing, Networking, Storage and Analysis*, SC 2019, (New York, NY), 2019.

[5] H. Rihani, P. Sanders, and R. Dementiev, "Multiqueues: Simple relaxed concurrent priority queues," in *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '15, (New York, NY, USA), p. 80–82, ACM, 2015.

[6] M. Wimmer, J. Gruber, J. L. Träff, and P. Tsigas, "The lock-free k-lsm relaxed priority queue," *SIGPLAN Not.*, vol. 50, p. 277–278, Jan. 2015.

[7] D. Alistarh, J. Kopinsky, J. Li, and N. Shavit, "The spraylist: A scalable relaxed priority queue," in *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP 2015, (New York, NY, USA), pp. 11–20, ACM, 2015.

[8] 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), pp. 209–221, Springer Berlin Heidelberg, 2015.

[9] D. Nguyen, A. Lenharth, and K. Pingali, "A lightweight infrastructure for graph analytics," in *ACM Symposium on Operating Systems Principles*, SOSP '13, (NY, USA), 2013.

[10] S. Yesil, A. Heidarshenas, A. Morrison, and J. Torrellas, "Understanding priority-based scheduling of graph algorithms on a shared-memory platform," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pp. 46:1–46:14, 2019.

[11] K. Pingali and et. al., "Ordered vs. unordered: A comparison of parallelism and work-efficiency in irregular algorithms," in *ACM Symposium on Principles and Practices of Parallel Programming*, PPoPP, 2011.

[12] M. A. Hassaan, D. Nguyen, and K. Pingali, "Kinetic dependence graphs," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pp. 457–471, 2015.

[13] D. Zhang, X. Ma, M. Thomson, and D. Chiou, "Minnow: Lightweight offload engines for worklist management and worklist-directed prefetching," *SIGPLAN Not.*, vol. 53, p. 593–607, Mar. 2018.

[14] M. C. Jeffrey, S. Subramanian, C. Yan, J. Emer, and D. Sanchez, "A scalable architecture for ordered parallelism," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 228–241, Dec 2015.

[15] M. Shan and O. Khan, "Accelerating concurrent priority scheduling using adaptive in-hardware task distribution in multicores," *IEEE Computer Architecture Letters*, vol. 20, no. 1, pp. 17–21, 2021.

[16] H. Dogan, M. Ahmad, J. Joao, and O. Khan, *Accelerating Synchronization in Graph Analytics using Moving Compute to Data Model on Tilera TILE-Gx72*. IEEE, 2018.

[17] H. Dogan, F. Hijaz, M. Ahmad, B. Kahne, P. Wilson, and O. Khan, "Accelerating graph and machine learning workloads using a shared memory multicore architecture with auxiliary support for in-hardware explicit messaging," in *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 254–264, May 2017.

[18] D. Sanchez, R. M. Yoo, and C. Kozyrakis, "Flexible architectural support for fine-grain scheduling," *SIGARCH Comput. Archit. News*, vol. 38, p. 311–322, Mar. 2010.

[19] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C. Miao, J. F. Brown III, and A. Agarwal, "On-chip interconnection architecture of the tile processor," *IEEE Micro*, vol. 27, no. 5, pp. 15–31, 2007.

[20] S. K. Moore, "Breaking the multicore bottleneck," October 2016. [Online; posted 28-October-2016].

[21] H. Dogan, M. Ahmad, B. Kahne, and O. Khan, "Accelerating synchronization using moving compute to data model at 1,000-core multicore scale," *ACM Trans. Archit. Code Optim.*, vol. 16, pp. 4:1–4:27, Feb. 2019.

[22] U. Meyer and P. Sanders, "delta-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003. 1998 European Symposium on Algorithms.

[23] L. H. O. Rios and L. Chaimowicz, "A survey and classification of a* based best-first heuristic search algorithms," in *Advances in Artificial Intelligence – SBIA 2010* (A. C. da Rocha Costa, R. M. Vicari, and F. Tonidandel, eds.), pp. 253–262, Springer Berlin Heidelberg, 2010.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[25] J. Nešetřil, E. Milková, and H. Nešetřilová, "Otakar borůvka on minimum spanning tree problem translation of both the 1926 papers, comments, history," *Discrete Mathematics*, vol. 233, no. 1, pp. 3 – 36, 2001. Czech and Slovak 2.

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

[27] J. J. Whang, A. Lenharth, I. S. Dhillon, and K. Pingali, "Scalable data-driven pagerank: Algorithms, system issues, and lessons learned.," in *Euro-Par* (J. L. Träff, S. Hunold, and F. Versaci, eds.), vol. 9233 of *Lecture Notes in Computer Science*, pp. 438–450, Springer, 2015.

[28] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, Dec. 2011.

[29] C. Demetrescu, A. V. Goldberg, and D. S. Johnson, eds., *The Shortest Path Problem, Proceedings of a DIMACS Workshop, Piscataway, New Jersey, USA, 2006*, DIMACS/AMS, 2009.

[30] J. Leskovec, K. Lang, A. Dasgupta, and M. Mahoney, "Community structure in large networks: Natural cluster sizes and the absence of large well-defined clusters," *Internet Mathematics*, vol. 6, 11 2008.

[31] L. Backstrom, D. Huttenlocher, J. Kleinberg, and X. Lan, "Group formation in large social networks: Membership, growth, and evolution," in *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, (New York, NY, USA), p. 44–54, Association for Computing Machinery, 2006.